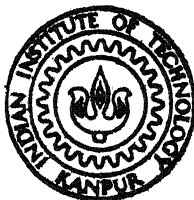


SPEECH PROCESSOR ARCHITECTURES AND SIMULATION OF TMS 32010

by
M. R. KESHEOREY



DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

APRIL, 1987

SPEECH PROCESSOR ARCHITECTURES AND SIMULATION OF TMS 32010

**A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of
MASTER OF TECHNOLOGY**

**by
M. R. KESHEOREY**

**to the
DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR**

APRIL, 1987

CONFIDENTIAL

Acc. No. **A 98316**

001.6425

lovingly
dedicated
to
My parents


Sri Ramchandra Shankarlal
and
Smt. Malati Ramchandra

11

CERTIFICATE

This is to certify that the thesis entitled, 'SPEECH PROCESSOR ARCHITECTURES AND SIMULATION OF TMS 32010' by Sri M.R. KESHEOREY has been carried out under my supervision and that it has not been submitted elsewhere for a degree.

April, 1987.


(A. Joshi)
Assistant Professor
Department of Electrical Engineering
Indian Institute of Technology
Kanpur 208016, INDIA

ACKNOWLEDGEMENTS

I would like to express my heartfelt thanks to

-Dr A. Joshi, my thesis supervisor, for his guidance and encouragement throughout this work,

-Padmasri Dr. R.P. Shenoy, Director, L.R.D.E., Bangalore and the Defence Research and Development Organisation for providing me with the opportunity to do this course,

-Mr. Nirmal Roberts for his help in the software part of this thesis,

-Mr. V.G. Rao, Mr. R. Subramanian and Mr. N. Sitaram my officers at L.R.D.E, Bangalore for constant encouragement throughout the course,

-my friends Mr. T. Jayakumar and V.S. Mahalingam at L.R.D.E., Bangalore for willingly taking care of my financial and administrative problems and supplying me the technical literature needed to complete this thesis,

-my wife Mrs. Chhaya and my sons Bharat and Abhay for their sacrifice in leaving me to my 'day and night' studies, and

-Mr. U.S. Mishra, for the excellent typing of this thesis.

-M.R. Kesheorey

CONTENTS

	<u>Page</u>
LIST OF TABLES	vii
LIST OF FIGURES	viii
ABSTRACT	x
CHAPTER 1 INTRODUCTION	1
REFERENCES	5
2 SPEECH PROCESSING	6
2.1 Introduction	6
2.2 Speech Production Mechanism	8
2.3 Speech Production Model	11
2.4 Speech Processing Systems	12
2.4.1 Speech Analysis Systems	12
2.4.2 Speech Synthesis Systems	14
2.4.3 Speech Analysis-Synthesis Systems: Vocoder	14
2.5 Linear Predictive Coding of Speech	16
2.6 Excitation	18
2.7 Hardware Requirements	18
2.8 Conclusion	20
REFERENCES	21
3 ARCHITECTURES FOR SPEECH PROCESSING	23
3.1 Introduction	23
3.1.1 Desirable Features of Speech Processors	24
3.1.2 An Ideal Speech Processor	27
3.2 Architecture 1	31
3.3 Architecture 2	33
3.4 Architecture 3	37
3.5 Architecture 4	39
3.6 Comment on Different Architectures	41
3.7 Architecture of TMS 32010	43
3.7.1 Arithmetic Elements	43
3.7.2 Data Memory and Auxiliary Registers	46
3.7.3 Program Counter and Stack	47
3.7.4 Peripherals and Program Memory	48
3.8 Bottlenecks of TMS 32010 as an Speech Processor	48
REFERENCES	51

CHAPTER 4	ASSEMBLER FOR TMS 32010	
4.1	Introduction	52
4.2	Executable Instruction Format	53
4.3	Pseudo-Operations	54
4.4	Assembly Process	55
4.5	How to Use the Assembler	60
4.6	Error Messages	62
4.7	Conclusion	63
	REFERENCES	64
5	SIMULATOR FOR TMS 32010	65
5.1	Introduction	65
5.1.1	TMS 320 - Programmer's Point of View	66
5.2	Simulation Procedure	74
5.3	Design Aspects	79
5.3.1	General	80
5.3.2	Integer Arithmetic	82
5.3.3	Interrupts	84
5.3.4	Load	85
5.3.5	I/O Ports	86
5.3.6	Plot	87
5.3.7	Trace	88
5.3.8	Deposit and Display	88
5.3.9	Break Points	89
5.4	Special Features	90
5.5	How to Use the Simulator	91
5.5.1	Deposit and Display Facilities	92
5.5.2	Trace Facilities	92
5.5.3	Break Point Facilities	94
5.5.4	Execution Facilities	97
5.5.5	Miscellaneous Facilities	98
5.5.6	Model Session	98
5.6	Conclusion	100
	REFERENCES	100
6	RESULTS	100
6.1	Introduction	100
6.2	Digital Filter Implementation	100
6.2.1	Filter Specifications	100
6.2.2	Design Equation	100
6.2.3	Flow Chart and Program Details	100
6.2.4	Results	100
6.3	FFT Program	100
6.3.1	Program Description and Flow Chart	100
6.3.2	Results	111
	Conclusion	111

	<u>Page</u>
CHAPTER 7 CONCLUSION	120
7.1 Introduction	120
7.2 Conclusion	120
7.3 Suggestions for Future Work	122
APPENDIX	
APPENDIX A Assembler Syntax	124
APPENDIX B Error Messages and Warnings	127
APPENDIX C Assembler Program Listings	130
APPENDIX D Simulator Program Listings	153

LIST OF TABLES

<u>Table</u>	<u>Title</u>	<u>Page</u>
5.1	Simulator Command Table	76
5.2	Mapping of TMS 320 Words to Simulator Words	82
6.1	Data Memory Allocation for FFT Computations	110

LIST OF FIGURES

<u>Figure</u>	<u>Title</u>	<u>Page</u>
2.1	Speech Production Mechanism	10
2.2	Source Filter Model of Speech Production	10
2.3	Speech Verification System	13
2.4	Speech Synthesis System	13
2.5	Channel Vocoder	15
2.6	LPC Vocoder	15
3.1	Architecture 1	32
3.2	Architecture 2	34
3.3	Architecture 3	38
3.4	Architecture 4	40
3.5	Block Diagram of TMS 32010	44
4.1	Assembler Flow Chart	56
5.1	TMS 32010 Status Register	71
5.2	Flow Chart of Simulation Steps	77
5.3	Block Diagram of TMS 32010 Simulation System	81
6.1	Digital Filter Program Flow Chart	104
6.2	Step Response	105
6.3	Impulse Response	1006
6.4	FFT Program Flow Chart	111
6.5	FFT of Real Symmetric Square Wave	116
6.6	FFT of Imaginary Asymmetric Points	117

LIST OF SYMBOLS

<u>SYMBOLS</u>	<u>Meaning</u>
' '	Text within Inverted commas is printed on the terminal by the system
' > '	Simulator prompt
>	Hexadecimal numbers are specified by the symbol before the number
< >	Items within angle brackets are defined by user
[]	Items within brackets are optional
{ }	Items within braces are alternative items, one of them must be entered.

ABSTRACT

Speech is a primary means of communication between people. Developments in the digital IC technology and the development of good digital signal processing algorithms has made processing the speech in real time feasible. With this development the speech signal processing activity has come to a stage from where low bit rate speech communication and man-machine communication via speech looks feasible. Whereas the speech processing activity in 1975-1985 period centered around bit - slice microprocessors using TTL technology, the current trend in speech signal processing is the use of special purpose signal processing chips. This thesis addresses itself to the discussion of architectures for speech processing.

The salient features of speech processing architectures reported in literature which are based on AM 2900, AM 2903 and MC 10800 bit-slice microprocessors are discussed. Simulation of Texas Instrument's Signal processing chip TMS 32010 has been done on DEC system 1090 computer at IIT Kanpur using a modular PASCAL program. A debugger incorporated as a special feature of the simulator has also been presented. A simple assembler for TMS 32010 has been written. Illustrative application programs such as a 17 tap antialiasing digital filter program and FFT program to compute 64 point complex DFT have been run on

the simulator. These programs confirm the suitability of TMS 32010 for speech processing. Bottlenecks of TMS 32010 architecture have also been pointed out.

CHAPTER 1

INTRODUCTION

Speech is the primary means of communication for human beings. Speech production mechanism suggests that if speech parameters are extracted from speech, they could provide additional advantages. The advantages are, low bit rate communication, man machine communication by voice, speaker identification, automatic speaker recognition to name a few. Miniaturization and the advent of large scale integrated circuits in electronics industry brought the microprocessors. Speech processing requires a large number of computations to be done. If the speech is to be processed in real time, the processor speed has to be about 4 to 5 MIPS. Bit slice microprocessors using TTL technology offered solution to the real time speech processing problem.

Different architectures of real time speech processor evolved from 1977. The VLSI technology available in 1980's has made signal processing chips for speech processing feasible. TMS 32010 is one of them. This thesis is concerned with investigation of different architectures for real time speech processing and simulation of the TMS 32010 with the facilities available at IIT Kanpur. For simulation, choice had to be made between the PDP 11 system available in Electrical Engineering Department and the DEC system 10 available at computer centre of IIT Kanpur. PDP 11 system was found unsuitable for the task because the basic

word length of PDP 11 system is 16 bit whereas the arithmetic section of TMS 32010 does 32 bit fixed point arithmetic. Expansion facility to enable the PDP 11 system to perform double precession arithmetic is not available at present. The software solution to this problem acted as a deterrent to use PDP 11 to simulate TMS 32010.

Having decided in favour of DEC System 10 computer for running the simulation program a further choice had to be made, between one of the several general-purpose high level languages (like PASCAL, FORTRAN) and specifically hardware simulation oriented languages like AHPL. It was decided to use a general-purpose high level language for the following reasons:

- 1) A special-purpose language would require learning afresh and the limited time available would not permit requisite depth,
- (2) Higher-level language provides portability between computers,

Among the high-level languages PASCAL has been chosen in preference to other languages because of the following reasons:

- (1) The inherent capacity of PASCAL for character handling and structured syntax simplifies the implementation,
- (2) Writing and checking of the program in PASCAL is easier,
- (3) Written description of the program more or less depicts the algorithm, whereas in FORTRAN just by looking at the

(4) Once the program is broken up into smaller modules, the way it is arranged in PASCAL is much better than in FORTRAN.

(5) PASCAL has better data structures; and thus the Memory and CPU registers of TMS 320 can be represented in a better way in PASCAL.

(6) The case statement in PASCAL is much better than for example, the assigned go to or computed go to statements in FORTRAN.

(7) PASCAL on most of the systems has better run time diagnostics which help in writing and running correct programs in FORTRAN even if a program works, it may not be giving the proper result or even the error might go un-noticed.

(8) In an environment like that of IIT Kanpur, where PASCAL is widely used by students, it will be easy for the user to understand the program and modify it suitably, if necessary, in future.

The thesis is organized into seven chapters including the current one. Above mentioned reasons give the motivation and the approach followed for this project.

Chapter 2 starts with basics of speech and speech production. A source filter model is explained for speech production. Speech processing systems are mentioned briefly. Basics of LPC channel vocoder are given from the view point of the number of computations. The chapter ends with the computational requirements of the real time speech processor and

suggestions about the speech processor architecture to achieve them.

Different speech processor architectures based on bit-slice microprocessors are given in Chapter 3. Improvements of one above the other are also discussed. The chapter ends with the architectural description of Texas Instrument's Signal processing chip TMS 32010 as a speech processor. The drawbacks of TMS 32010 are also listed out.

Chapter 4 describes the design of the assembler for TMS 32010. The facilities provided in the assembler and its operation are also given.

Chapter 5 gives the simulation details for TMS 32010 simulator. The debugging facilities provided as special features of the simulator are also explained. The methodology adopted for simulation is given.

Chapter 6 gives the results of a digital filter program and FFT program run as bench mark programs. In addition to verification of the assembler and simulator by these programs, the suitability of TMS 32010 chip for speech processing application problems is also established.

Chapter 7 gives the conclusion and lists few suggestions for future work. TMS 32010 assembly language syntax, Assembler and simulator error messages, and program listings are given in Appendices A, B and C respectively.

REFERENCES

- [1] J.L. Flanagan, "Speech Analysis Synthesis and Perception", 2nd Ed., Springer-Verlag, 1972.
- [2] J.L. Flanagan, "Speech Coding", IEEE Trans. on Communications, Vol. COM - 27, No. 4, April 1979, pp. 710-733.
- [3] K. Jensen and N. Wirth, "PASCAL User Manual and Report", 3rd Ed., Narosa Publishing House, New Delhi, 1985.

CHAPTER 2

SPEECH PROCESSING

2.1 INTRODUCTION

Speech is by far the most common method of communication amongst human beings. Speech is a peculiarly human activity, not endowed to other species. There is no doubt that animals, birds, and even insects can communicate within their species using well understood but fixed sets of sounds. What they lack is a communication system with the flexibility of speech and the ability to string together patterns of sounds to signify different and new things. In the beginning man was not very different from other species in the use of vocal organs, most of the communication being done by symbols using hands. Invention of speech was not due to the need to express his thoughts but due to the difficulty of "talking with his hands full". He found his hands too busy in the work and learnt to use the voice organs to produce speech for communication [1].

At the acoustic level, speech signals consist of fluctuations in the air pressure which propagate to an acoustic receiver, the human ear. Sensitivity of the ear is limited and the acoustic energy of the speech signal diminishes rapidly with distance. Hence acoustic wave is not a good means of distant communication. Until the invention of telephone by Alexander Graham Bell, speech as such was not used for long distance

communication. The telephone acts as a transducer which varies the intensity of electric current precisely as the air varies density during the production of speech sounds. Bandwidth of conversational telephone channel is about 3 KHz with an SNR of 30 db. Channel capacity is given by Shannons formula [14]

$$C = BW \log_2 (H^S/N) \text{ bits/sec} \quad (2.1)$$

Equation (2.1) indicates that a normal telephone channel has the capacity to transmit information at rates 30,000 bits/sec.

The information that is communicated through speech is of discrete nature; i.e. it can be represented by a concatenation of elements from a finite set of symbols. These discrete symbols (elements) from which every sound can be classified are called phonemes. Each language has its own distinctive set of phonemes, typically numbering between 30 and 50. English language, for example has got 42 phonemes. Limit on the rate of physical motion of speech articulators requires that the humans produce speech at an average rate of about 10 phonemes/sec. If the phonemes are represented by a set of 6 bit binary number ($2^6 = 64 < 50$), average information rate of speech comes out as $10 \times 6 = 60$ bits/sec. In other words the written equivalent of speech contains information equivalent to 60 bits/sec. at normal speaking rates. If the correlation between pairs of adjacent phonemes is taken into consideration,

Different experiments [1] to measure the capacity of human channel indicates that human being is not capable of processing information at rates greater than 50 bits/sec. This information rate for written equivalent of speech is about 600 times lower than the capacity of a normal telephone channel. It suggests the need to "process" the speech signal to bridge this large gap between the average information rate of speech and the channel capacity to transmit it. It can be argued that human speech contains much more information than the written equivalent such as the emotional state of the talker. However, such a large gap between the two capacities still remains unjustified. The speech processing has to make use of the constraints characterizing the production and perception of speech.

2.2 SPEECH PRODUCTION MECHANISM [1], [9], [10]

Basic elements of human speech production mechanism are shown in Fig. 2.1. It consists of vocal-tract which is a non-uniform acoustic tube, terminated by the lips at one end and by the vocal cord constriction at the other end. The shape of the vocal tract (and hence its acoustic property) continuously changes during speech production by voluntary movement of the articulators; namely, the lips, jaws and tongue. The nasal-tract begins at the velum and terminates at the nostrils. The vocal-tract is coupled to or decoupled from the nasal-tract by controlling the movement of velum. For nasal sounds, Velum is open and sound is radiated from both the mouth and the nostrils

For the production of non-nasal sounds the velum is drawn tightly up and it effectively seals off the entrance to the nasal cavity.

These passive cavities (vocal tract/nasal tract) are excited by forcing air from the lungs in trachea and through the glottis. The 'voiced sounds' of speech are produced by vibrating action of vocal-cords. The period of vibration of vocal cords is known as 'pitch period'. The resulting air pressure through vibrating vocal cords is quasiperiodic and excites the vocal tract to produce voiced sounds. Another source of vocal excitation is acoustic noise due to turbulence of air created at a narrow constriction in the vocal tract. This results in the production of 'unvoiced sounds'. Voice pitch frequency is talker dependent. It varies typically between 50 to 100 Hz for men and between 100-400 Hz for women and children.

Most basic property of the speech waveform is that they are band limited. Voiced speech segments are characterized by high energy, quasiperiodicity and less number of zero-crossings. The unvoiced speech segments are characterized by relatively low signal level, noise like appearance and more number of zerocrossings. It is usually difficult to mark the boundary between the voiced segments and the unvoiced segments.

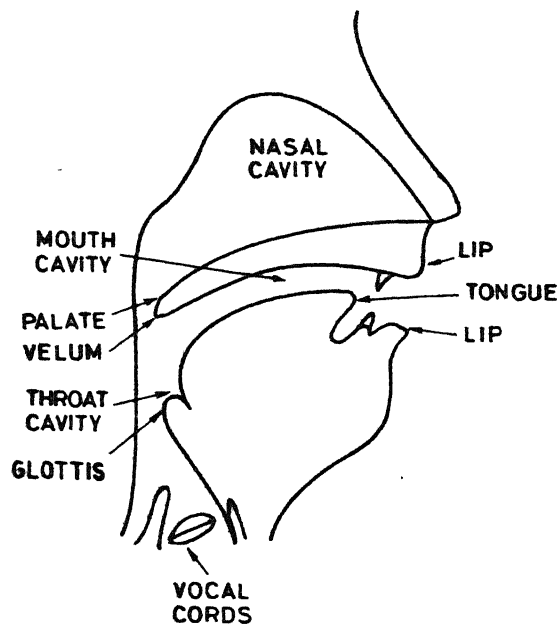


FIG.2.1 SPEECH PRODUCTION MECHANISM

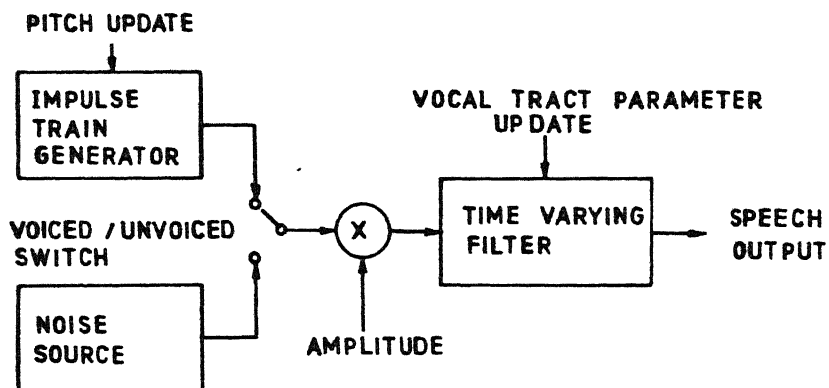


FIG. 2.2 SOURCE FILTER MODEL OF SPEECH PRODUCTION

2.3 SPEECH PRODUCTION MODEL [1]

Sound source and the vocal tract are two distinct physical entities. The vocal tract changes shape rather slowly in speech production. The voiced and unvoiced excitations are mutually exclusive. Therefore, a very simple source filter model as shown in Fig. 2.2 can be used to represent the speech production mechanism.

The vocal tract is represented by the time varying filter. Effect of nasal cavity is to introduce additional poles. The time varying filter takes care of those additional poles. Excitation source is either a quasiperiodic impulse train or a noise generator. Voiced/unvoiced switch selects one of them according to the voiced-unvoiced nature of the speech. The amplitude control regulates the energy output. Silence can be represented by zero energy. The parameters for the vocal tract filter, voiced/ unvoiced switch, pitch period and the amplitude are regularly updated so as to keep track of the variations in speech waveform. These parameters vary very slowly and an update at every 10-20 msec. is sufficient. Study of the speech production mechanism and its representation as a simple source-filter model suggests the possibility of coding speech information in forms other than merely the transduced pressure wave.

2.4 SPEECH PROCESSING SYSTEMS

Speech processing systems can generally be divided into the following three classes [2], [3] :

1. Speech analysis systems
2. Speech synthesis systems
3. Speech analysis - synthesis systems.

2.4.1 Speech Analysis Systems [2]

In speech analysis systems, input speech is processed to get one or more parameter of the source-filter model. An appropriate action is then taken based on those parameters. Examples of speech analysis systems are - voice response systems, speaker verification system and speaker identification system.

Figure 2.3 shows a ^{block} ~~biode~~ diagram of an on line speaker verification system. The person wishing to be verified first enters his claimed identity. Then he utters his verification phrase; and requests some action to be taken; in the event he is verified.

The sample utterance which occurs some where within a preselected time period is first accurately pinpointed. This is done by "end point detection" system. Once the beginning and end of the utterance have been found; speech is analysed to give a series of parameters. For example, the pitch detector is used to measure the pitch contour of the

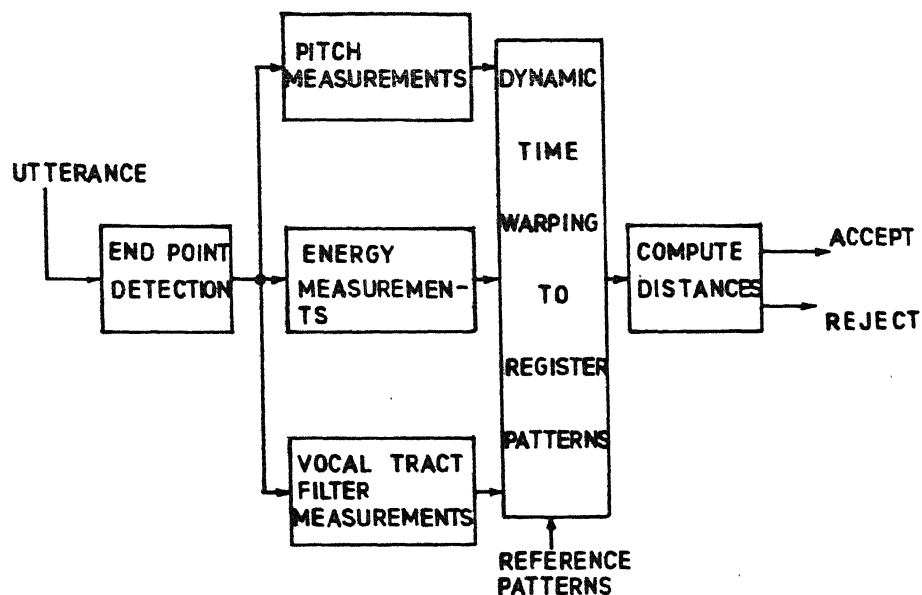


FIG. 2.3 SPEECH VERIFICATION SYSTEM

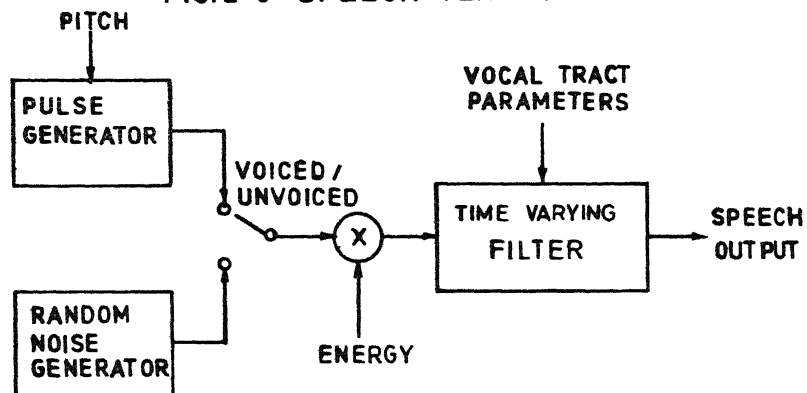


FIG. 2.4 SPEECH SYNTHESIS SYSTEM

utterance. Dynamic comparisons with the stored parameters decide whether to accept or reject the talker.

2.4.2 Speech Synthesis Systems [2],[6]

Speech processing systems in which speech is synthesised using the stored speech parameters are known as speech synthesis systems. Examples of speech synthesis systems are talking toys, talking calculators etc. Figure 2.4 shows the block diagram of a typical speech synthesis system. The excitation sources are a variable frequency pulse generator for voiced speech frames and a random noise generator for unvoiced speech frames. The amplitude of the selected source is multiplied by the energy level and then is applied to the filter. The filter is a time varying digital filter whose filter coefficients are programmed as per the filter coefficients of the speech frame.

2.4.3 Speech Analysis - Synthesis Systems : Vocoder

Vocoders are a class of speech processing systems in which speech is first analyzed to extract the speech parameters. These parameters are then used to synthesize the speech. These systems are popularly known as vocoders. In order to implement a vocoder, various types of techniques are in general possible both for vocal tract filter identification and for excitation identification. The channel vocoder, the formant vocoder, the LPC vocoder and the homomorphic vocoder [1], [2], [3], [4] are different vocoders based on different

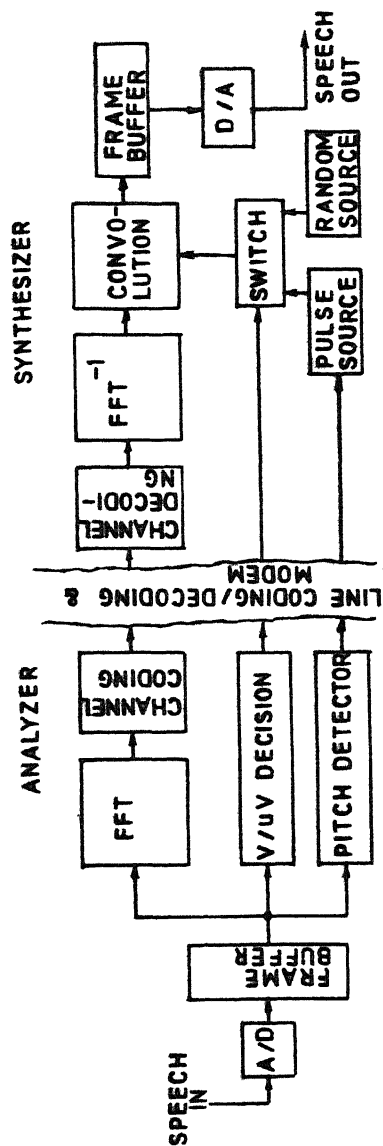
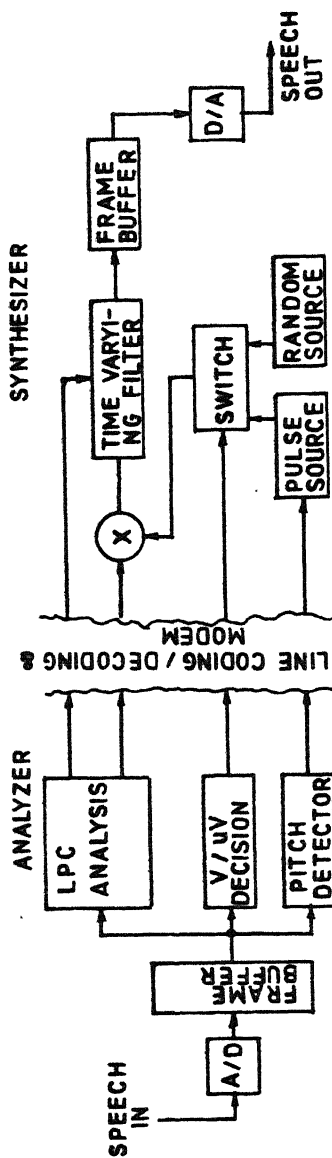


FIG. 2.5 CHANNEL VOCODER



schemes of vocal tract filter modelling.

In channel vocoder a suitably chosen filter bank in analysis side extracts short-time energy for every acoustic band of interest. This information feeds the synthesizer side as shown in Fig. 2.5.

In LPC vocoder some coefficients related to the overall shape of vocal tract filter characteristics are identified. Block diagram of LPC vocoder is shown in Fig. 2.6. Since it is well documented in the text [5] a cursory discussion of linear predictive coding of speech is given in the next section.

2.5 LINEAR PREDICTIVE CODING OF SPEECH

s_n , the sampled speech signal is predicted by a linear combination of last p samples. The predicted speech sample \hat{s}_n is given by

$$\hat{s}_n = - \sum_{k=1}^p a_k s_{n-k} \quad (2.2)$$

The error between the true s_n and the predicted \hat{s}_n is the prediction error e_n

$$e_n = s_n - \hat{s}_n \quad (2.3)$$

Total squared error E over a block $N_1 \leq n \leq N_2$ is

$$E = \sum_{n=N_1}^{N_2} e_n^2 \quad (2.4)$$

Set of predictor coefficients which give minimum

squared error are the desired predictor coefficients. That is, a set of simultaneous equations given by (2.5) are to be solved to get the coefficients a_i

$$\frac{\partial E}{\partial a_i} = 0 \quad \text{for } 1 \leq i \leq p \tag{2.5}$$

The solution to above problem is [4] ,

$$\sum_{k=1}^p a_k \sum_{n=N_1}^{N_2} s_{nk} \cdot s_{ni} = - \sum_{n=N_1}^{N_2} s_n \cdot s_{n-1} \quad 1 \leq i \leq p \tag{2.6}$$

Depending on the interval over which the error is minimized, whether it is of infinite duration ($-\infty < n < \infty$) or of finite duration ($0 \leq n \leq N-1$) there are two methods, the autocorrelation method or the covariance method. In autocorrelation method the coefficient matrix is Toeplitz and in covariance method the coefficient matrix is symmetric. Recursive methods to solve set of equations (2.6) are available [4].

Once the coefficients a_i are solved, their values along with the excitation parameters are transmitted to the receiver. At the receiver speech synthesis is done using the speech synthesizer.

2.6 EXCITATION

Knowledge of excitation parameter is necessary in speech analysis and speech analysis - synthesis systems. Usual methods for voiced - unvoiced decision are energy and average zero crossing rate of the signal. Pitch determination methods also usually give the ^{cl}ue whether an analysis frame is voiced or not. There are several pitch period estimation techniques. They are explained and compared in [11] and [12].

2.7 HARDWARE REQUIREMENTS:

(a) Flexibility:

There are a number of speech processing algorithms and they must be tested on the processor to evaluate the relative performance. This necessitates that the processor must be programmable.

(b) Real Time Capability:

A typical LPC vocoder needs 12×10^3 operations (add and/or multiply) to process 22.5 msec. speech [13]. That is, a real time processor has to perform 5×10^5 operations per second. One operation needs several machine instructions, depending on the architecture. Usually processor which can perform four to five million instructions per second (MIPS) is capable of processing speech in real time. Specific hardware requirements of the processor from real time processing capability are:

- 1) it should be efficient in instruction execution
- 2) to provide enough efficient sequencing instructions
- 3) to provide instructions for fast integer arithmetic
- 4) to manipulate efficiently the structured data
- 5) to perform efficiently the input and output of the data for reduction of both time overhead and hardware complexity.

The speed requirements of 4 to 5 MIPS is met by choosing the processor architecture based on bit-slice μp 's using TTL technology. Separation between data memory and program memory as well as the large width of the microcode (usually more than 40 bit wide) ensures the first requirement. A good sequencer meets the second requirement. It gives powerful branch instructions (including call and return) and sufficient capability of nesting subroutines. The third requirement is fulfilled by choosing a separate multiplier preferably with an internal accumulator. The multiply or multiply-accumulate time of the multiplier being of the order of one or two clock cycles of the processor. The fourth requirement needs that the data memory or memories could be addressed in various ways as (i) direct addressing (ii) indirect addressing (iii) auto-decrement addressing and (iv) immediate addressing. The fifth requirement needs that the processor should be able to handle multiple interrupts generated by I/O devices and the I/O devices must have direct memory access.

2.8 CONCLUSION

Speech can be represented by source filter model. Based on this model there are various methods of speech processing. Linear predictive coding is the most popular method. To process the speech in real time, the processor speed should be four to five MIPS. The processor also needs a good program sequencer, separate program memory and data memory sections, a separate multiplier and the I/O devices having direct memory access. The processor must have good memory addressing capability and preferably multiple data memories.

REFERENCES

- [1] J.L. Flanagan, "Speech Analysis Synthesis and Perception", 2nd Ed., Springer-Verlag 1972, pp. 9-10, pp.7.
- [2] L.R. Rabiner et al., "Digital Processing of Speech Signals", Prentice Hall, Inc., Englewood Cliffs, NJ 1978.
- [3] A.V. Oppenheim, "Applications of Digital Signal Processing, Prentice Hall, Inc., Englewood Cliffs, NJ 1978.
- [4] J. Makhoul, "Linear Prediction : A Tutorial Review", IEEE Proc. Vol. 63 April 1975, pp. 561-580.
- [5] J. Markel and A. Gray, "Linear Prediction of Speech", Springer Verlag, 1976.
- [6] R.W. Schafer and J.D. Markel, "Speech Analysis", IEEE Press, 1978.
- [7] I.H. Witten, "Principles of Computer Speech", Academic Press, 1982.
- [8] L.M. Koehler et al. "Speech Output for HP Series 80 Personal Computers", Hewlett, Packard Journal, Jan. 1984, pp. 29-35.
- [9] "Speech Signal Processing", Speech Research Group, Department of ECE, IISc Bangalore, India, April 1977.
- [10] V.G. Rao et al. "Speech Encoding : State of Art Report", LRDE Publication No. R-2310, LRDE Bangalore, Feb. 1977.
- [11] L.R. Rabiner et al., "A Comparative Performance Study of Several Pitch Detection Algorithms", IEEE Trans. ASSP, Vol. 24, No. 5, Oct. 1976, pp. 399-417.
- [12] C.A. McGonegal et al. "A Subjective Evaluation of Pitch Detection Methods Using LPC Synthesized Speech", IEEE Trans. ASSP., Vol. 25, No. 3, June 1977, pp. 221-229.
- [13] P. Castellino et al. "Architecture and Algorithm Choices for an LPC Vocoder Implementation", ALTA FREQUENZA, Vol. LIII, No. 1, Jan. - Feb. 1984, pp. 24-36.

- [14] M. Schwartz, "Information Transmission, Modulation, and Noise", 2nd Ed., McGraw-Hill, Kogakusha Ltd., 1970, pp. 609.

CHAPTER 3

ARCHITECTURES FOR SPEECH PROCESSING

3.1 INTRODUCTION

In this chapter architectures for speech processors are discussed. Desirable features of speech processors are listed in Section 3.1.1. Features of an ideal speech processor which efficiently meets the speech processing requirement are brought out in Section 3.1.2. Four illustrative architectures using bit-slice microprocessors and the architecture of VLSI signal processor TMS 32010 are discussed. Section 3.2 describes architecture 1 which is based on AM 2901 bit-slice microprocessor and is a forerunner of other architectures. Architecture 2 given in Section 3.3 is based on AM 2903 bit-slice microprocessor which is an improvement over AM 2901. Architecture 2 improves upon architecture 1 not only on the CPU count but also on the addressing mode and access to the multiplier. Architecture 3, given in Section 3.4 is based on Motorola's M10800 MECL processor and is an example from ECL family of bit-slice microprocessors. Architecture 4 is a definite improvement on architectures 1, 2 and 3. A good amount of thought process has undergone to make it near ideal for speech processing applications. It uses multiplier - accumulator instead of the simple multiplier used in earlier architectures. It has different addressing modes. Architecture 4 is described in Section 3.5. Section 3.6

gives comments on architectures 1 to 4. The VLSI signal processor TMS 32010 given in Section 3.7 has most of the advantages of architecture 4 and comes nearest to the ideal speech processor. Weak points of TMS 32010 architecture as a speech processor are listed at the end of the chapter in Section 3.8.

3.1.1 Desirable Features of Speech Processors

Speech represented in the digital form has to be processed by the processor to extract its essential parameters. These parameters can be used to synthesize the speech at the receiving end. Speech processing algorithms such as FFT, LPC and convolutions [8] need computations which are structured. Multiplication, shifting and adding are most extensively used in these algorithms. Desirable features of a speech processor are discussed below:

- 1) Word Length: A basic word length of 16 bit is required for speech processing. A lower word length would fail to extract the essential features of speech. The reasons are (a) speech signal of 10 to 12 bits is needed to provide enough dynamic range for parameter extraction and (b) the truncation errors due to fixed word length distort the result considerably thereby making the parameter extraction impossible. A higher word length does not contribute to enhance the output of the system.

2) Arithmetic: Intermediate calculations such as sum of products, in speech processing have to be carried out with 32 bit precision. Hence an ALU of 32 bit or a facility to do double precision arithmetic is a desirable feature of speech processors.

3) Separate Multiplier: Speech processing algorithms need a large number of multiplications. Hence a separate multiplier to give a 32 bit product in one or two machine cycles is an essential feature of any speech processor. If the ALU is 16 bit, then a multiplier accumulator which can do a 16x16 product and add the result to a previously accumulated sum in a single machine cycle becomes a desirable feature of the speech processor.

4) Separate Data and Program Memory: Separation of data memory and program memory thereby providing two separate buses, provides instruction fetch and instruction execution to be done simultaneously. This improves upon the speed of the processor which becomes the slower of the two. Since in speech processing, speed of the processor is of prime concern, the practice of separating data and program memory has become a standard.

5) Sequencer: A sequencer providing good branch and jump capability provides an efficient way to run the programs which are quite structured. Another desirable feature of the speech processor is to have a sequencer with enough stack depth of provide nested subroutines. This greatly helps in developing speech processing packages by separate teams.

23

6) Peripherals: Speech usually is the input and output of the speech processing system. Hence, an analog to digital converter and a digital to analog converter are needed as peripherals. A 6 to 8 bit nonlinear A/D-D/A is sufficient for waveform coding of speech where exact replica of the speech waveform is achieved. However since parameters are extracted in speech processing systems; a linear A/D and D/A of 10 to 12 bits accuracy is needed. The sampling rate of low pass filtered speech is 8 to 10 KHz. The speech processor is interfaced with the MODEM on the line side with the help of serial to parallel converter and parallel to serial converter. Usual bit rate on line is 2.4 kbps and below. At sampling frequency of 10 KHz and modem data rate at 2.4 kbps, for a typical frame of 20 msec. there are 200 samples/frame on A/D - D/A side and 48 bits/frame on S/P - P/S side.

7) Data Memory: Data memory should have a direct access to the ALU. During speech processing not only the raw data takes part in computation but also various constants (such as sin/cos tables and log values) are frequently used. Therefore, a desirable feature of the speech processor is to have a data memory which is partly RAM and partly ROM and has direct access to CPU (DMA capability).

8) Addressing Modes: Structured data take part in speech processing. Therefore flexible addressing modes to access data are desirable for speech processors.

9) Program Memory: For speech processors, program resides in the program memory. If the processor is micro-processor based then for speed considerations horizontality of the microcode is a desirable feature. No decoding is done on the program memory out put and the program memory out put goes directly to different points to control various sections of the processor. This does not apply to VLSI based speech processors. Physical limitations on the number of pins of the processor make it necessary to have a smaller program memory word which is internally decoded by the processor to provide control signals to different sections.

10) Speed: Real time processing of speech puts up a minimum limit on the processor speed. The process^{or} has to complete the computations pertaining to a frame before next speech frame arrives. As discussed in Section^{2.7}, real-time speech processing involves 4 to 5 Million Instructions Per Second. The desirable features of a speech processor from speed point of view must be seen in connection with its architecture and its instruction set. A processor capable of running at speeds of 10 MIPS or above may not be a desirable one, compared to a processor capable of running at speeds just the half and having all the features listed above.

3.1.2 An Ideal Speech Processor

An ideal speech processor is required to analyze/synthesize speech in real time. It not only meets the

requirements listed in 3.1.1 but also performs required computations efficiently. This helps in two ways - (a) processor needs lesser number of cycles to execute a program and (b) the user can write the programs more efficiently - conserving the program memory space and his own efforts. Architectural details of an ideal processor are described below. It is felt that with the current technology such a processor can be realized.

1) CPU: Basic word length of the ideal processor need not be more than 16 bits. However all the arithmetic is to be performed with 32 bit accuracy. A barrel shifter to convert a 16 bit data to 32 bit data is a part of the CPU. A 16x16 multiplier giving 32 bit product in one cycle and outputting the result to the 32 bit ALU can be as effective as a separate MAC (Multiplier - Accumulator) with 16 bit ALU. CPU also has to have a sufficiently large scratch pad; with good addressing capability. A scratch pad of length $256 + 16$ would be ideal. A length of 256 is sufficient to store a block of data required either for LPC implementation or for FFT calculations. The extra 16 words could be used to store infrequently used but important parameters. The addressing of the scratch pad should be direct addressing or by one of the two separate registers with autoincrement/auto decrement facilities. The two registers would provide the capability of performing indexed array type calculations in an efficient manner.

In addition to the scratch pad described, a 1Kx16 data

memory is a must for the ideal speech processor. Non-immediate data and constants reside in the data memory. Data memory is directly accessible to CPU. Data or constants stored in the data memory can directly go to the ALU or the multiplier. Auto increment addressing facility to the data memory would provide transfer of a block of data or constants in the scratch pad which has even more flexible addressing modes.

2) Sequencer: Sequencer of an ideal speech processor would have the branch instructions depending on the accumulator status ($= 0$, < 0 , $< > 0$, ≤ 0 , > 0 , ≥ 0), a branch on zero depending on the address register selected for the scratch pad (providing indexing and branching simultaneously), call and return instructions and vectored interrupts. A 12 bit word deep stack is felt ideal. This would provide the facility to develop the software in independent modules. Vectored interrupt facility cuts down the hardware and software overheads for data acquisition from peripherals.

3) Program Memory: A 4kx16 memory devoted to program is felt necessary for the ideal processor. Though a program memory of 2kx16 is sufficient for a speech processor, a 4kx16 memory provides the added advantage of the program to be developed in modules by independent teams. The price paid in terms of the access time and the memory space requirements is negligible compared to the advantage of better utility. Horizontality of the microcode which needs a large program -

memory has no room in the ideal processor.

4) Peripheral Devices: 12-bit A/D - D/A and 16 bit S/P - P/S directly hooked on the data bus with direct access to the CPU are essential for the ideal processor. The direct memory access to the peripherals on interrupts saves the overhead to interact with the peripherals. 16 bit S/P and P/S conversion reduces the number of interrupts for S/P and P/S devices as compared to 8 bit S/P and P/S. Since the data bus is 16 bit and S/P and P/S are directly hooked on the data bus, ideal thing is to get 16 bits from or supply 16 bits to the processor at a time.

5) Speed: 5 to 6 MIPS speed would enable the ideal processor to have additional features. Some of them as nonhorizontality of microcode, 4K of program memory, and auto-incrementing, autodecrementing facility with a branch on scratch pad address register equal to zero have already been discussed. Other useful feature that could be provided by choosing a slower speed is to provide powerful instructions such as an instruction that moves data in the scratch pad to the next location, loads the data in the multiplier register, accumulates the multiplier and autoincrements/auto decrements the scratch pad address registers in a single machine cycle. This is what exactly is done in a typical digital filter. Since LPC synthesis is nothing but a digital filter excited by voiced/unvoiced pulses, the speech synthesis time can

greatly be reduced by such instruction. A moderate speed of 5 to 6 MIPS also does not put great constraint on the Program memory and data memory selection.

3.2 ARCHITECTURE 1

The basic block^{ock} diagram of architecture 1 given by E.M. Hofstetter et al., 1 is shown in Fig. 3.1. The nucleus of this system is the CPE which is based on the AM 2901 [2] microprocessor chip. All instructions are executed in a 150 ns cycle except the multiply which requires four machine cycles. However, using the currently available multiplier chips, the multiplication time can be brought down to one cycle. It is a 16 bit processor and has two separate buses. The IBUS and the OBUS. The IBUS is multiplexed between 6 data sources, the 12 bit A/D converter, the 8 bit serial to parallel converter, the 16 bit memory output register (MOR), the 16 bit upper and lower products coming from the multiplier and an 11 bit field coming from the instruction register. The data memory consists of 2K-16 bit words, 1.5K of which are ROM and contain various look up tables. The I-bus acts as an input to the CPE. The output of the CPE is channeled to one of the six registers. These are D/A converter, parallel to serial (P/S converter, the memory buffer and memory address registers (MBR and MAR), and the multiplicand and multiplier (MCD and MPR), registers of the multiplier.

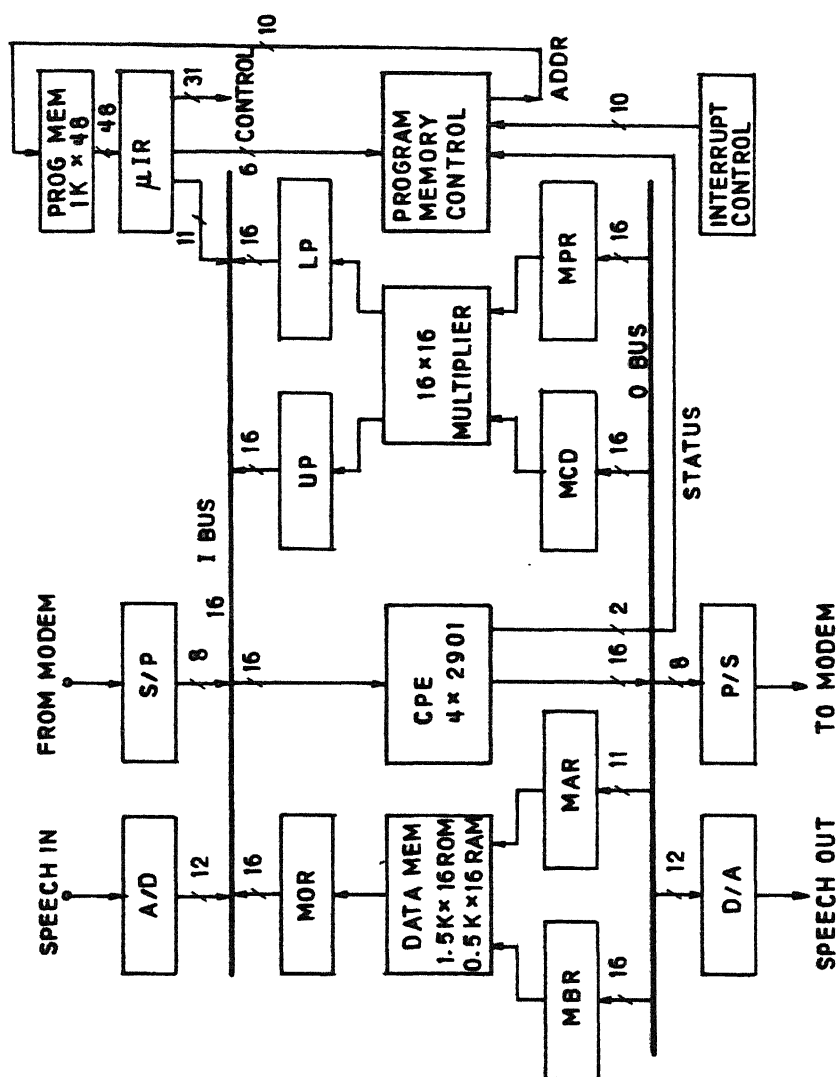


FIG. 3.1 ARCHITECTURE 1

Data memory and program memory are separate. Program memory is 1K x 48 bits. This allows parallel operations of the control section (accessing next microinstruction) and of the execution section, (performing calculations). Output of the program memory is clocked into a microinstruction register. Program memory address is derived from the program control logic. This is based on the AM 2909 [2] program sequencer chip. The 2909 controller is driven by a 2 bit control line which selects the next program memory address from one of the four addresses. These addresses are last address plus one (continue), the latest address on the internal stack (return), an interrupt address (Jump to interrupt) determined by the I/O system and a jump address which comes from the microinstruction register (conditional or unconditional branch). The jump logic thus allows for unconditional jumps. Conditional jumps, depending on the status bits coming from the CPE and jumps to and returns from subroutines. As the internal stack on AM 2909 is four word deep, the subroutines may be nested up to four deep when interrupts are locked out and three deep when they are active. Reference [3] describes a processor built on similar lines in more details.

3.3 ARCHITECTURE 2

The block diagram of architecture 2 given by B.C. Shin et al [4] is shown in Fig. 3.2. It has 16 bit CPU and all the

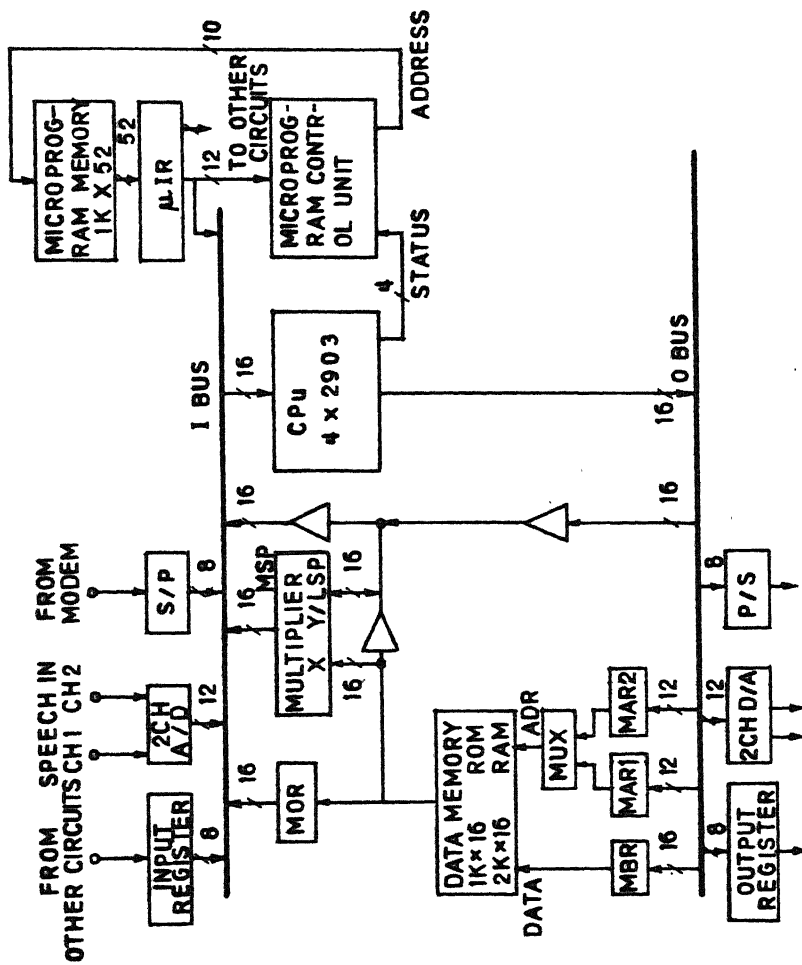


FIG. 3.2 ARCHITECTURE 2

instructions are executed in a cycle time of 208 ns. The main sections of the processor part are data memory, central processing unit (CPU), a multiplier and microprogram control unit (MCU). The CPU is interfaced with the rest of the system via two buses, an input data bus (IBUS) and an output data bus (OBUS). The nucleus of the system is the CPU which uses AM 2903 [2] bipolar microprocessor chips. AM 2903 has special functions such as division and double length normalization as compared to AM 2901. The I bus is a unidirectional tristate bus and is connected to one of seven data sources. These are - the input register, the A/D and serial to parallel (S/P) converters, the micro-instruction register (MIR), the memory output register (MOR) and upper and lower products coming from the multiplier. The output of the CPU is channeled through the OBUS to the D/A and parallel to serial (P/S) converters, two memory address (MAR1 and MAR2) registers, the memory buffer register (MBR), the Y input of the multiplier and the output register.

The data memory is composed of 2K RAM and 1K ROM. The RAM is used for data buffering and temporary storage. The ROM stores various lookup tables. To enhance the data processing capability, pipe line technique is used in the data transfer between the CPU and the data memory. That is, old data can be read out while the new data memory address is being setup. The two address registers MAR1 and MAR2 have the

autoincrement facility. The address multiplexing and the automicrementing is controlled by the control signal coming from the microinstruction register. The memory output can be fed directly to the X and Y inputs of the multiplier, without passing through the CPU. The default value of the address multiplexer comes from MAR1. MAR2 is chosen only when the memory output data are routed to the Y input of the multiplier. Therefore this scheme facilitates the altering access of two block data sets in different memory locations without additional addressing from the CPU. Accordingly, it is easy to copy or transfer a data set from one location to another, and to calculate the sum of products from the two data sets $\{X_i\}$ and $\{Y_i\}$ obtained in autocorrelation calculations. Addressing modes possible are direct, indirect, indexed and autoincrement.

The multiplier has the multiplication time less than one machine cycle. Since the input and output of the multiplier are fully buffered pipeline technique may be used for efficient multiplication (as used for accessing data memory).

Microprogram memory is 1Kx53. The output of this memory is clocked into a microinstruction register and then executed. AM 2910 [2] program sequencer is used. AM 2910 has 9 word deep stack. Thus a maximum of 9 nested subroutine calls can be made.

3.4 ARCHITECTURE 3

Block diagram of the architecture 3 given by M. McLaughlin et al [5] is given in Fig. 3.3. It is a 16 bit, fixed point processor and uses Motorola's 4 bit-slice M 10800 MCL processor family parts. Whole of the system uses MCL logic except a hardware multiplier. The basic cycle time for the processor is 90 nsec, with a multiply requiring 3 cycles or 270 nsec.

The ALU (4-MC 10800's) performs arithmetic, logical and shift operations. Communication among the different parts of the processor is achieved via three buses labelled as the O-Bus, A-Bus and I-Bus. The O-Bus and I-Bus are bidirectional. In general the I-Bus and A-Bus serve as inputs to the different sections in the arithmetic and data memory sections and the O-Bus functions as an output bus. The register file consists of 16 general purpose 16 bit registers. The registers can function as accumulators, as pointers or as index registers. Outputs of the register file serve as inputs to the ALU, memory interface and multiplier. The file can read two locations simultaneously. During one cycle, two registers can be read and either one of those two can be written into. This gives more flexibility than the A and B registers contained in AM 2901 and AM 2903.

Data Memory is 4Kx16. The I/O ports are addressed as a specific memory location in data memory. Address and

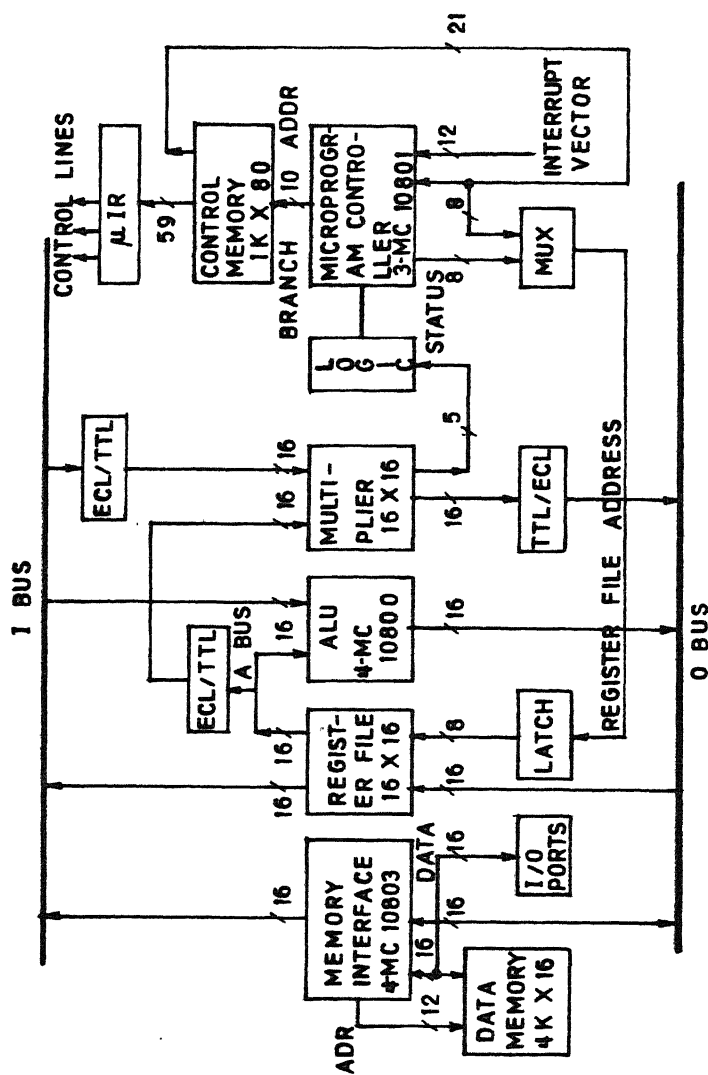


FIG. 3.3 ARCHITECTURE 3

data to the data memory and I/O port are obtained through the memory interface block. The control memory is 1Kx80. Bits from the control memory go to the microprogram controller and to the micro-instruction register. Program memory address is generated by Microprogram Controller (3-MC 10801's). The processor has 7 interrupt lines on a prioritized basis. When an interrupt is received, control is automatically transferred to one of the seven vectors in the program memory. The condition code register is a status register which decides the conditional branches in the sequencer.

3.5 ARCHITECTURE 4

Architecture 4 given by V. Castellino et al. [6] is shown in Fig. 3.4. Its micro-instruction speed is 325 ns. It has two subsections which allow maximum parallelism. These sections are execution section and sequencing section. The execution section consists of (a) data handling subsection and (b) address handling subsection. Since this architecture has four separate buses (Y, ADR, A and B), there is more flexibility to data flow. The data memories RAMA and RAMB are of 1Kx16 each. Both these memories are provided with separate address registers. ADRA and ADRB. ADRA and ADRB are hooked on the ADR bus. The ADR bus is driven either by the ALU output or by the sequencer circuit. This provides direct addressing, indirect addressing, autoincrement addressing and immediate

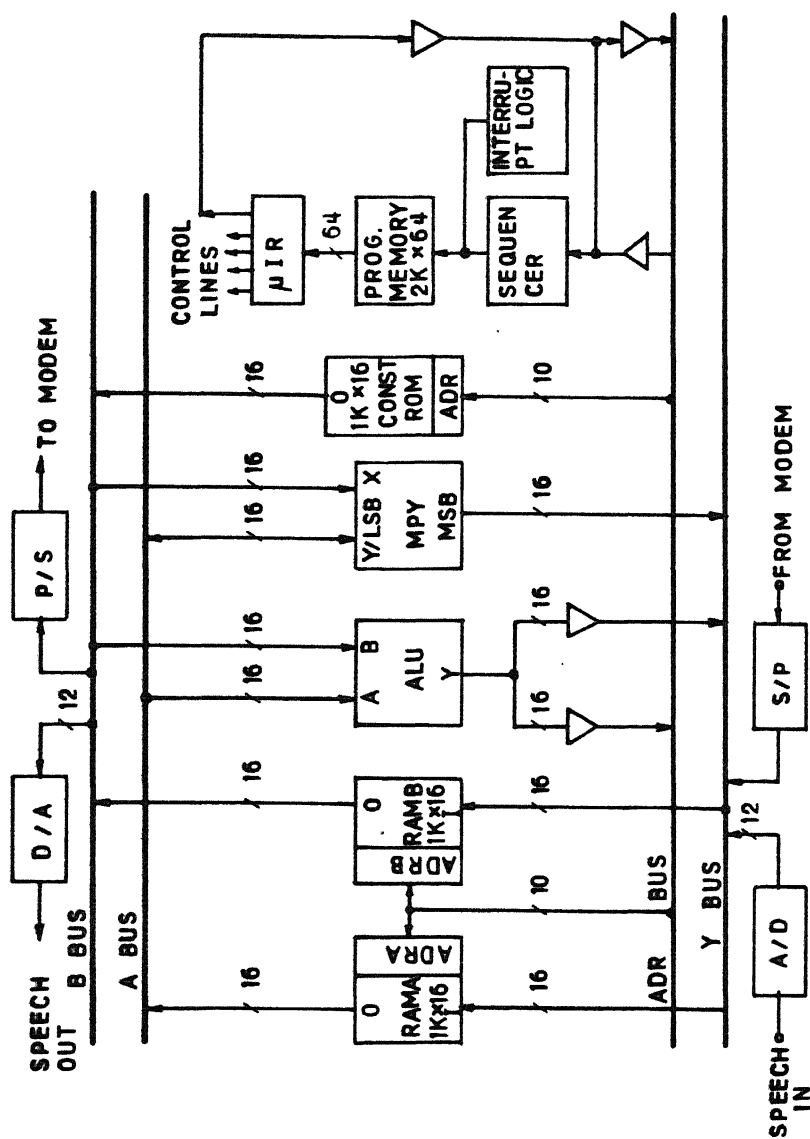


FIG. 3.4 ARCHITECTURE 4

addressing to both the RAMs. Another noteworthy feature of this architecture is the use of ^Mmultiplier, - Accumulator chip instead of a simple multiplier. This enables the execution of operations of the kind $\sum_{i=1}^n a_i b_i$ with ease. RAMA and RAMB can be loaded with the starting addresses of vectors; a_i and b_i . Then n products and n accumulations give the final result. Another feature of this architecture is a separate constants ROM which is used for boot's trapping initial data into the RAM. Constant Memory is $1K \times 16$ bits.

The sequencing section is based on AM 2910 and has all the advantages already discussed in Section 3.3. The micro-program memory is $2K \times 64$ bits. 12 bit A/D and D/A converters and serial to parallel and parallel to serial converters act as interrupts to the processor.

3.6 CURRENT AND DIFFERENT ARCHITECTURES

The architectures discussed in previous sections span over a period of 7 years (1977 to 1984) in which the digital IC technology had undergone a lot of change. There has not been any significant development in the basic algorithm except a better understanding of the LPC algorithm by experience due to lot of experimentation. All the hardware architectures discussed so far have implemented LPC algorithm in one form or the other though, they are all programmable and as such any speech processing algorithm can be implemented on them.

Architecture 1 is the forerunner of speech processing architectures using bit-slice microprocessors and has established the fact that real-time speech processing systems can be implemented. Use of separate program memory and a separate data memory; use of more than one data bus and the use of a separate multiplier are the salient points of architecture 1. The instruction cycle time with the present day technology may come down below 100 ns.

Architecture 2 improvised upon architecture 1 by using AM 2903 instead of AM 2901 as the CPU and using AM 2910 instead of 2909 as sequencer. Straight away it helped in better data handling capability and better program flexibility. However, the main feature of this architecture is the use of two address registers (though multiplexed) to address the data memory and the connections of the multiplier. This arrangement results in efficient calculations of sum of products, which is quite often used in speech processing.

The architecture 3 is not much different from Architectures 1 and 2. The 16x6 bit register file gives slightly better data handling capability than the A and B registers (16x16 bit) contained in AM 2901 and AM 2903. However, this advantage is offsetted by 80 bit wide program memory. In the absence of the data sheets for the chips used in this processor not much comment can be made.

The architecture 4 comes up with many new ideas. Use of multiplier-accumulator chip makes the computation of sum of products even simpler. The two address registers which have the

capability of autoincrement and can act upon two arrays makes the architecture very powerful. The third new idea was the use of a separate constants ROM which can be loaded into the data memory when desired to perform array type calculations. Use of AM 2910 as the sequencer gives nested looping upto 9 level deep. This is certainly much better than the facility provided by a 4 word deep stack in architecture 1. The speed of the processor is 325 nsec per micro-instruction with respect to the component technology of 1979. Speeds of the order less than 200 nsec are possible using the current technology.

3.7 ARCHITECTURE OF TMS 32010

Block diagram of Texas Instruments Signal Processing Chip TMS 32010 which is very much suitable for real time speech processing applications is shown in Fig. 3.5. TMS 32010 has two separate buses, the data bus and the program bus. Both are 16 bit wide. Since the program memory and data memory are separate, a full overlap of instruction fetch and execution is possible. System clock cycle is 200 ns and most of the instructions need one clock cycle for execution. Details of the TMS 32010 architecture are given below.

3.7.1 Arithmetic Elements

The ALU, the accumulator, the multiplier and the shifters are main basic arithmetic elements. All arithmetic operations are performed using two's complement arithmetic.

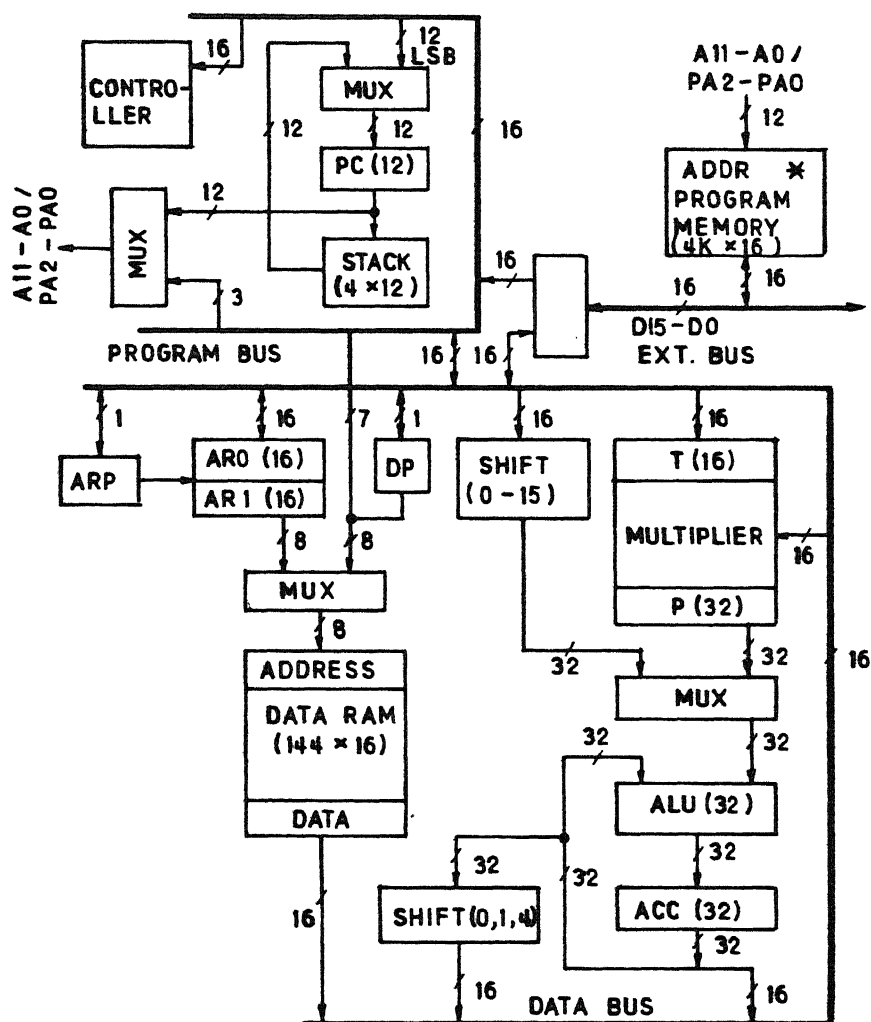


FIG. 3.5 BLOCK DIAGRAM OF TMS 32010

The ALU can perform arithmetic and logical operations. Arithmetic operations are performed on 32 bit word, whereas the logical operations are performed on 16 bit word. While performing logical operations, the upper 16 bits of the accumulator are forced to zero. When the data to the ALU is from the data memory and arithmetic operation is to be performed, Shifter (0-15) comes into picture. Input to the accumulator is thus a 32 bit word with full sign extension and with given shifts with respect to the addressed data memory word. 32 bit wide accumulator can be stored in two words of the data memory. The lower 16 bits of the accumulator can be stored separately. A shift of 0, 1 or 4 can be given while storing the upper 16 bits of the accumulator. Though the store takes place with the shift, accumulator remains unchanged.

The 16x16 bit multiplier gives the 32 bit product and makes it available in T register. The multiplier output is the product of the contents of the T register and the data on the data bus, addressed from the data memory. Provision is also there to multiply the T register contents with a 13 bit signed constant. Instructions are available to multiply and accumulate. Thus calculation of sum of products can efficiently be done.

The processor has a provision to alter the result if an overflow occurs. A control bit called OVM (overflow mode) bit

can be set or reset by software instructions. If an overflow occurs when OVM is set, the most positive or the most negative representable value of the ALU is loaded into the accumulator. When an overflow occurs when OVM bit is reset, the accumulator is unmodified. OVM mode is equivalent to clipping. This facility is very useful in speech processing.

The accumulator overflow is stored in a separate bit which forms the part of status register. Various types of branches on the accumulator status (<0 , ≤ 0 , >0 , ≥ 0 , $<>0$, and $= 0$) are possible.

3.7.2 Data Memory and Auxiliary Registers

Data memory consists of 144 words of 16 bit. All non-immediate data operands reside in data memory. Provision is there to load data into data memory from program memory and vice versa by TBLR and TBLW instructions. Data can also be read in the data memory from peripheral devices. Contents of a data memory location can likewise be transferred to a peripheral device. Provision is there to connect upto 8 input/output devices to the external bus.

Data memory can be addressed in a number of ways. Address to the data memory is either from the array registers, ARO and AR1 or from the microinstruction itself. DP is a data page pointer which can be loaded with 1 bit constant. DP is a part of status word. In the direct addressing mode, the address to the data memory is given by DP and the microinstruction word bits 0 to 6. Page 0 contains data memory

locations 0 to 127 and page 1 contains data memory locations 128 to 143. In the indirect addressing mode the address to the data memory is either from ARO or from AR1. ARO and AR1 are 16 bit registers which can be read or written. ARP is array register pointer and forms the part of status word. It can be read or written by the system. The lower nine bits of the array registers act as an up down counter. The ARP selects the array register for indirect addressing. If ARP is 0 then lower 8 bits of ARO provide address to the data memory. While performing an operation using indirect addressing, the concerned array register can be either incremented or decremented and the ARP can be changed. This provides the facility of indexed array. Branch Instruction **BANZ** which allows the branch when auxiliary register is not zero is also helpful in providing loops in the program. The auxiliary register can also be used as temporary storage locations.

3.7.3 Program Counter and Stack

The instruction fetch part of the processor contains the program counter (PC) and the stack. Program memory is always addressed by the contents of the PC. Stack is four word deep. Program counter usually points to the one higher value of the last PC value except in the cases of jumps, returns, call and interrupts. For the returns, the address is supplied by the top of the stack. For servicing the interrupt

the PC is set to 2. Any jump or call address is supplied by the program bus to the PC.

3.6.4 ⁷Peripherals and Program Memory

TMS 32010 has a single interrupt which always points to ^{second} some location in the program memory. Added to it there is a $\overline{\text{BIO}}$ pin which when zero causes a jump on instruction BIOZ . Thus the status of a peripheral can be monitored at $\overline{\text{BIO}}$ pin and a suitable action can be taken. During the IN and OUT instructions the processor supplies the page address across A11-A0/PA2-PA0 lines. At that time the external bus is released so that data transfer takes place via external bus. A total of 8 input and 8 output devices can be hooked on the external bus.

TMS 32010 supports 4K external program memory addressed by A11-A0/PA2-PA0 lines. The program memory in usual practice is a RAM/ROM combination. Since data can be read from and written on the program memory, I/O operations can be performed by the program memory itself. Memories with access time less than 100ns are recommended for the TMS 32010 processor.

3.8 Drawbacks of TMS 32010 as a Speech Processor

TMS 32010 provides many good features of an ideal speech processor. However some of the drawbacks are listed below

1) In-sufficient data memory: Since the CPU has direct access only to the data memory, it is not possible to interact with the peripheral devices without any overhead. The speech applications in which a block of speech data is to be processed (say for example FFT of 256 real samples) this overhead becomes too much. Provision is there in the software to expand the internal data memory upto 8 pages. Internal data memory of the order of 1K is ideal for speech processing and TMS 320's later versions would not have this handicap.

2) Insufficient stack depth: In real time speech processing a four deep stack is usually insufficient. TBLR and TBLW operations practically limit the stack available in TMS 32010 to a depth of three. In speech processing applications it is not uncommon that different subtasks are handled by different group of people. For example a vocoder system can be divided into three subtasks: (1) analysis of vocal tract filter parameters, (2) analysis of excitation parameters and (3) synthesis of speech. In a situation like this, managing with a four deep stack becomes difficult and the user has to be much careful while using the stack. There is a provision to expand by software the stack into data memory. However, it again needs overhead which can be ill-afforded in real time speech processing. One wishes that in

future, a chip with TMS 32010 architecture is available with more stack.

3) Poor interaction with I/O devices: With TMS 32010 user has all the facilities once the signal is inside the data memory. Since in TMS 32010's architecture CPU cannot access any outside data, the overhead time to acquire data is too large.

REFERENCES

- [1] E.M. Hofstetter et al., "Microprocessor Realization of a Linear Predictive Vocoder", IEEE Trans. Acoust. Speech Signal Processing, Vol. ASSP-25, No.5, Oct. 1977, pp. 379-387.
- [2] Advanced Micro Devices, "Bipolar Microprocessor Logic and Interface", AM 2900 family 1983 data book.
- [3] V.G. Rao et al., "Programmable Signal Processor", Electro Technology, Vol. 25, No. 4, Dec. 1981, pp. 160-165.
- [4] B.C. Shin et al., "Implementation of a Two-Channel LPC Vocoder", IEEE Trans. Commun., Vol. COM-31, No.7, July 1983, pp. 907-915.
- [5] M. McLaughlin et al., "High Performance Processor for Real-Time Speech Applications", ICASSP-80, pp. 859-863.
- [6] P. Castellino et al., "Architecture and Algorithm Choices for an LPC Vocoder Implementation", ALTA FREQUENZA, Vol. LIII - No. 1, Jan.-Feb. 1984, pp. 24-36.
- [7] Texas Instruments, "TM 32010 User's Guide", Nov. 1983, Revision A.
- [8] L.R. Rabiner, et al., Digital Processing of Speech Signals", Prentice-Hall, Inc., New Jersey, 1978.

CHAPTER 4

ASSEMBLER FOR TMS 32010

4.1 INTRODUCTION

In this chapter the design aspects of an assembler for TMS 32010 are outlined. Function of the assembler is to translate a source program written in the assembly language to the machine language. Basic aim of this assembler is to produce object code for TMS 32010 efficiently. Modular programming technique has been followed while writing the assembler. Adequate documentation is provided for an interested user to understand the assembly process. In Section 4.1 format of executable instructions is given. Section 4.2 explains the pseudo-operations provided in this assembler. The assembly process is discussed in details in section 4.4. Section 4.5 shows how to use the assembler. Error messages given by the assembler show the exact nature of the error in the source program, and thus help the user to write the correct program efficiently. These are explained in Section 4.6. Section 4.7 gives the conclusion. All the assembly language instructions of TMS 320 have been implemented and tested carefully.

4.2 EXECUTABLE INSTRUCTION FORMAT

The TMS 320 assembly language is line oriented. Each source statement is a combination of the following four fields:

- (1) Label
- (2) Instruction Mnemonic
- (3) Operand
- (4) Comment.

These fields are described in detail in the following.

The source file statements must be unnumbered. The label field is the first field of the source line and starts at the first column of the source line. Fixed format is followed for labels. All labels are of two characters. First character of the label is an alphabet and the second character of the label is a number. Label is terminated by a colon.

The instruction mnemonic field follows the label field and is separated from it by at least one blank. When there is no label the instruction field must not start before the fourth column of the source line.

The operand field follows the mnemonic field and is separated from it by at least one blank. When two or more operands appear in a statement they must be separated by at least one blank and must not contain embedded spaces. In TMS 320 assembly language some of the operands are optional. Exactly

the same amount of flexibility is provided in this assembler as given in the user's manual [1]. Operand field for the branch and call instructions must be a symbol having the same syntax as that of the label stated earlier. The numbers in the operand field are decimal numbers. When negative numbers are encountered (as in MPYK or in ZLQ instructions) a "-" sign followed by blank is put before the number.

The last field in any statement is the comment field. Comments are strings of characters which are inserted in the source program to identify or clarify the individual statements or program flow. Comment field is optional^{on}. It is separated by the assembler syntax expression by at least one blank and starts with a semicolon. This field is ignored by the assembler but is included in the listing. Appendix A shows the format of an instruction.

4.3 PSEUDO-OPERATIONS

ORG and ZEQ pseudo-ops are provided. Pseudo-ops cannot be labelled. Pseudo-ops appear at the place of instruction mnemonic of the executable instruction. They have the operand field and also have optional^{on} comment field. Pseudo-ops are not executed by the machine but are interpreted as follows.

ORG pseudo-op specifies the program location counter value from where the following entries in the assembly language source program are loaded. No default value of the program counter is provided. Operand of ORG pseudo-op is the decimal value of the program counter from where the following source program is to be loaded.

Besides storing the program, the program memory of TMS 320 is also used to store constants, look up tables and data values. ZEQ pseudo-op provides the facility to load these values. Operand field of the ZEQ pseudo-op is a string of decimal numbers which are to be loaded in the subsequent locations of the program memory. The numbers in the string are separated by atleast one blank.

4.4 ASSEMBLY PROCESS

Format of the assembly language syntax is same as the format given in [1]. This is given in Appendix A as an aid to the interested user. Pseudo-ops provided in this assembler are also listed in Appendix A. The special symbols used in this appendix are listed after Table of Contents.

Figure 4.1 shows the assembler flow-chart. ERRX, $0 \leq X \leq F$ is the error message by the assembler. Error messages are listed in Appendix B. Appendix B also gives a typical assembler listing with errors in the source program.

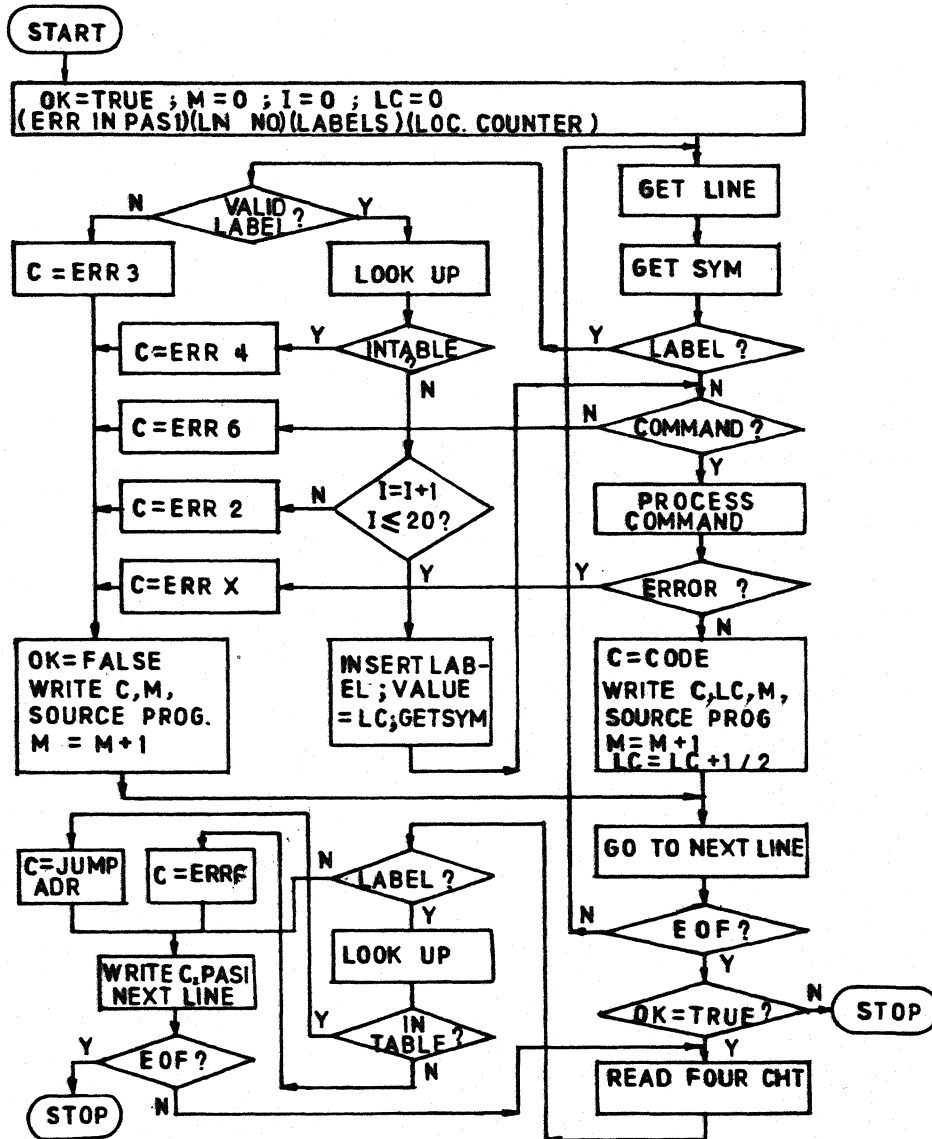


FIG. 4-1 ASSEMBLER FLOW CHART

Figure 4.1 shows the flow-chart of the Two Pass assembler. Functions done in Pass One are:

- 1) Reading a line of the source program.
- 2) Processing the line - Processing includes generation of label table by the label field, coding of the instruction, including the branch instructions having backward references.
- 3) Writing the code/error message/forward reference in PAS1 file and copying the source program.

Functions done in Pass Two are:

- 1) Filling up forward referenced branch addresses.
- 2) Entering the complete PAS1 file including source program in user defined file.

"OK" is an error flag which is reset if any error occurs during Pass One. M and I shown in Fig. 4.1 are variables used for the source program line number and number of labels in the label table respectively. "LC" is the program location counter. Care has been taken to develop the program in modules. One line, terminated by CR, of the source program is processed at a time. Procedure "GETLINE" copies a line of the source program. Procedure "GETSYM" scans the internally stored line till a nonblank character is encountered. After this it starts copying the information in ^{an} ~~err~~ array (SYM) till the next blank is encountered. This procedure is called repeatedly while processing a line. The information

can be a label, a mnemonic, a pseudo-op, an operand or an end of line. Beginning of the comment field or CR, whichever occurs first is treated as end of line and is excluded from the assembly process.

If the symbol happens to be a valid label then it is searched in the label table by procedure "LOOKUP". Label table is defined as an array (LTBL) of two dimensions. If the symbol is found then same symbol has been used twice. Therefore an error message is flashed on the terminal and further assembly of that line is aborted, after registering the error message in PAS1 file. If the symbol is not found then it is entered in the last location of the label table as a new entry by procedure "INSERT". The size of the label table is incremented by one and the value of the label is set to the current LC value. Linear search of the label table is carried out and hence a restriction of 20 labels is imposed on the user to keep the assembly time reasonable.

The symbol next to the label or the very first symbol in the line in the absence of a label is a mnemonic or a pseudo-operation. Mnemonic, and pseudo-ops are declared as a record Y. This record is arranged in the alphabetical order. A binary search of this record is made. A total of 62 mnemonics and pseudo-ops are present in the assembler. A worst case search of 6 is needed to pinpoint a particular mnemonic or a pseudo-op. A case statement then directs the flow of control to a definite procedure. The use of binary search and the use of case statement reduces

the assembly time. Each mnemonic or pseudo-op is processed by a separate procedure whose name is identical to the mnemonic or pseudo-op with a prefix P. This gives better readability to the assembler program. For example, mnemonic ADD is processed by procedure PADD.

Attempt has been made to encode an instruction as much as possible during Pass One itself. The branch instructions which have backward references are also fully encoded. Branch instructions with forward references put the symbolic address preceded by * in place of object code.

Appendix B contains the Pass One listing with an example of forward referencing. This is fitted in during Pass Two when all the labels are declared. The assembler creates a file called PAS1 during Pass One. As soon as the processing of a line is over the program writes the object code in PAS1 file. If however an error is encountered while processing a line, error code ERPX, $0 \leq X \leq F$, is put in place of the object code (APPB). Along with the object code or the error code (as the case may be) PLC value in hex and line number in decimal are recorded. Full line as written in the source program is then copied after the PLC value in PAS1 file. Pass One continues to encode the source program till the program ends, even if an error has been encountered in some line.

If Pass One is error free then execution of Pass Two is carried out. During Pass Two only the object code position in PAS1 file is read. If a "*" symbol at the first column and

a blank in the fourth column is met, search for the label given by the second and third column is made in the label table. If the label is found then the value corresponding to the label is entered in object file. Entries of PASS1 file along with the Pass Two modifications are entered in the user defined object file.

4. HOW TO USE THE ASSEMBLER

The user has to run the assembler program to start the assembly process. The assembler prompts the user to provide the input file name of upto six characters as -

'GIVE INPUT FILE NAME : XXXXXX

When the user has provided the name of his input file the assembler reads it in the packed array NINP [1] - NINP [6]. If the file name happens to be of less than six characters, the NINP [1] - NINP [6] array is left justified and added with blanks on the right. A standard extension for the input file is INP. This name is then attached to the internal file name "FINP" by the assembler. Assembler then prompts the user to provide the output file name of upto six characters as -

'GIVE OUTPUT FILE NAME : XXXXXX

When the user has provided the name of his output file the assembler reads it in the packed array NOUP [1] - NOUP [6]. If the file name happens to be of less than six characters, the NOUP [1] - NOUP [6] array is left justified and added with blanks on the right. A standard extension for the

output file is OBJ. This name is then attached to the internal file name "FOUP" by the assembler.

Standard PASCAL provides facility to open the text files [4]. Text files are files that consist of a sequence of characters that are subdivided into variable length lines. The predefined type "TEXT" is used to declare text files.

Handling of the text files is very much similar to the handling of INPUT and OUTPUT files. PASCAL statements for these are given below:

WRITE (F, Ch)	- Write a character variable in text file F.
READ (F, Ch)	- Read a character variable in text file F.
WRITELN (F)	- Terminate the current line of text file F.
READLN (F)	- Skip to the beginning of the next line of the text file F.
EOLN (F)	- a Boolean function indicating whether the end of the current line in the text file F has been reached.
RESET (F)	- Initiate inspection (reading) of F.
REWRITE (F)	- Initiate generation (writing) of F.

The PASCAL compiler available on DEC system-10 at IIT Kanpur [5] has extended forms of RESET and REWRITE statements. It provides the facility to set up correspondence between the file designators and actual files. Statements used for this purpose used in the assembler to define the input file and the output file are:

RESET (F, file name) : F corresponds to file name, initiate inspection of F.

REWRITE (F, file name) : F corresponds to file name initiate generation of F.

Other extensions to RESET and REWRITE statements are not used and hence the files inspected and generated are restricted to user's area only.

After the user has provided the input and output file names, the assembler assembles the source program provided by the input file. Errors during Pass One are flashed on the terminal. If no errors are encountered in Pass One, then pass two is carried out automatically. If any error is encountered during Pass Two the message is again flashed on the terminal.

Assembler output is given in two files. PAS1 file which is created by the assembler for intermediate operations is available to the user for inspection. This file is useful for the correction of errors in the source program. Final output is in the user defined file with an extension OBJ. In case any errors are detected in the source program the user has to correct them and rerun the assembler.

4.6 ERROR MESSAGES

The assembler not only does the basic function of translating the assembly language source program into machine language program but also points to specific errors in the

source program. Since the assembly of a line is aborted once an error is met, the error indicated by the assembler is the very first error in that line. Assembler obviously points out the syntax errors only. Erroneous lines in the source program are highlighted by adding "XXX" characters after the error message. Errors can be recorded by seeing the terminal while assembler is assembling the program or they can be read from the PAS1 file created by the assembler. Error messages and their meaning is listed in Appendix B.

4.7 CONCLUSION

The assembler produces correct object code for all valid instructions of TMS 320. It is a very basic assembler but at the same time is good enough to enable the user to write application programs to be run on the simulator. Size of the label table is fixed but can easily be changed to incorporate larger number of labels in the source program. Adequate documentation is provided to make the user understand the methodology adopted during assembling. Any further additions to the mnemonics (due to future developments in TMS 320) can easily be carried out as the structure of the program is modular. For further details user is advised to refer to the user's manual for TMS 320 [1].

REFERENCES

- [1] Texas Instruments, "TMS 32010 User's Guide", Revision A Nov. 1983.
- [2] D.W. Barron, "Assemblers and Loaders", Third Edition, Macdonald and Jane's, London, 1977.
- [3] B. Dash, "M 68000 Simulation System", M.Tech. Thesis, C.S. Department, IIT Kanpur.
- [4] K. Jensen and N. Wirth, "PASCAL User Manual and Report", Third edition, Narosa Publishing House, New Delhi.
- [5] R. Brooks, "Using PASCAL on DEC system - 10", Department of Information and Computer Science, University of California - Irvine, revised version April 8, 1976, p. 48-58 (PASREL.HLP on DEC System-10).

CHAPTER 5

SIMULATOR FOR TMS 32010

5.1 INTRODUCTION

Simulation is a very widely used technique. However, simulation of a complex processor as a tool for software development is of recent origin. Advantages of the simulator are ease of software development, testing and modifications. Main aim of the simulator for TMS 32010 is to provide the user at IIT Kanpur a facility to develop speech processing algorithms, test them and modify them. User has almost all the freedom for software development as if he were working with a H/W using TMS 320 processor. The simulator program uses the TMS 32010 object code developed by the assembler described in Chapter 4. Input and output files^{es} may be associated with the port address 0 of the I/O instructions in order to simulate I/O device which will be connected to the processor. The interrupt flag can be set periodically at a user defined interval for simulating an interrupt signal. Before initiating the program execution, break points may be defined and the trace mode may be set up. During program execution, the internal registers and memory of the simulated TMS 32010 are modified as each instruction is interpreted by the computer. Execution is suspended when either a break point or an error is encountered. Once program execution is suspended, the internal registers, and both the program and data memories can be inspected and/or modified. Provision is

made to load the program, constants and data in the program memory using the output of the assembler. A plot facility is provided to plot the contents of the program memory from one location to another clocation. The plot can be seen either on the terminal or a hard copy can be obtained on the printer. Section 5.1.1 gives the details of TMS 320 from a programmer's point of view. Addressing modes of TMS 32010, various instructions of TMS 32010 and care to be taken while performing certain instruction is also brought out. The function of \overline{BIO} , INTERRUPT and RESET pins of TMS 32010 is explained. Section 5.2 gives the simulation procedure. It outlines the steps taken in the simulation program. Design aspects of the simulation system are given in Section 5.3. Section 5.3.1 through 5.3.9 contain the general design aspects, integer arithmetic, design aspects for interrupts, load, I/O ports, plot, trace, break points and deposit and display facilities. Section 5.4 enumerates the special features of the simulator. Section 5.5 gives in details the method of using the simulation system. Section 5.6 concludes the chapter with conclusion.

5.1.1 TMS 320-Programmer's Point of View

TMS 320 is a signal processing VLSI chip. It has powerful instruction set, flexible addressing modes and other facilities (such as clipping the ALU result to positive or negative maximum). TMS 32010's salient points from a programmer's view point are described below. For details, the user is referred to [1].

Addressing Modes - Three main addressing modes are available with the TMS 32010 instruction set-direct, indirect, and immediate addressing. In direct addressing, seven bits of the instruction word concatenated with the data page pointer (DP) form the data memory address. Indirect addressing forms the data memory address from the least significant eight bits of one of two auxiliary registers, ARO and AR1. The auxiliary register pointer (ARP) selects the current auxiliary register. The auxiliary registers can be automatically incremented or decremented in parallel with the execution of any indirect instruction. In immediate addressing data are derived from part of the instruction word. Depending on the situation data is 1 bit, 8 bit or 13 bit. Examples of these addressing modes are shown below:

1) ADD 9 5 -

Add to accumulator the contents of memory location 9 left-shifted by 5 bits (DP = 0 assumed).

2) ADD *+ 8 1 -

Add to accumulator the contents of data memory address defined by contents of current auxiliary register (AR). This data is left-shifted 8 bit before adding. The current auxiliary register is autoincremented by 1. Auxiliary register pointer is loaded with a value 1 after execution (i.e., ARP = 1).

3) LARK ARO K 0 \leq K \leq 255 -

ARO is loaded with constant K.

Accumulator Instructions - Since the accumulator of TMS 32010 is 32 bit and the data bus is 16 bit, ^wWide variety of accumulator instructions are available. A shift code is associated with the

ADD, SUB and LAC instructions which specifies the number of shifts given to the contents of the data memory before forming a 32 bit word. The shift is an arithmetic shift and sign extension is provided. Logical operations such as AND, OR, XOR are available. Logical operations are performed with the lower 16 bits of the accumulator. Higher 16 bits of accumulator are forced to zero while doing logical operations. One can add to or subtract from higher order bits of accumulator with ADDH and SUBH instruction. 16 bit arithmetic can also be performed using ADDS and SUBS instructions. The result of an arithmetic operation which is usually 32 bit can be stored in a variety of ways such as store low order or high order accumulator bits (SACL, SACH). High order accumulator bits can be stored with a shift of 0, 1 or 4 bits. Conditional subtract (SUBC) instruction is useful for division.

Branch Instructions - Various branch instructions to provide branching depending on the accumulator condition are provided in TMS 320. These are BLZ (< 0), BLEZ (≤ 0), BGZ (> 0), BGEZ (≥ 0), BNZ ($\neq 0$), and BZ ($= 0$). Unconditional branch (B), branch on over flow (BV) and branch on auxiliary register not zero (BANZ) are other branches. Branch on over flow, apart from branching on over flow, resets the over flow flag. Over flow flag, when set, cannot be reset by any other method except by the BV instruction. Even a RESET signal does not alter the overflow flag. BANZ instruction decrements the current auxiliary register by 1 each time the branching takes place. CALL and RETURN are the usual instruction to call to or return from a subroutine.

Branch instructions are executed in 2 machine cycles and occupy two word of program memory.

Multiplier Instructions - The 16x16 - bit pipe lined multiplier consists of three units: the 16 bit T register (TR). The 32 bit P register (PR) and the multiplier array. In order to use the multiplier, the multiplicand must first be loaded into TR from data RAM by using one of the following instructions LT, LTA or LTD. Then the MPY (multiply) or the MPYK (multiply immediate) instruction is executed. If the MPYK instruction is used the multiplier value is a 13-bit constant derived from the MPYK instruction word. This 13-bit constant is right justified and sign extended. After execution of the MPY or MPYK instruction, the product will be found in PR. This product can be added to, subtracted from, or loaded into the accumulator by executing one of the following instructions: APAC, SPAC, LTA, LTD or PAC.

There is no way to restore the contents of the P register without altering other registers. For this reason in the TMS 32010, an interrupt is delayed until the instruction following the MPY or MPYK instruction has been executed. Thus, the MPY or MPYK instruction should always be followed by instructions that combine the PR with the accumulator: PAC, APAC, SPAC, LTA or LTD. This must always be followed as a logical consequence of the TMS 32010 instruction set.

I/O and Data Memory Instructions - Contents of data memory location can be copied into next location by DMOV instruction. Input and output of data to and from a peripheral is accomplished by the

IN and OUT instruction. Data is transferred over the 16 bit data bus to and from the data memory in two machine cycles. 8 input ports and 8 output ports are available with TMS 32010. The three multiplexed LSBS of the address bus, PA2 through PA0 (Fig. 3.5), are used as a port address by the IN and OUT instructions. The remaining higher order bits of the address bus, A~~11~~¹¹ through A3, are held at logic zero during execution of IN and OUT instructions.

The TBLR and TBLW instructions allow words to be transferred between program and data spaces. TBLR is used to read words from program ROM/RAM into the data RAM. TBLW is used to write words from on-chip data RAM to off-chip program RAM. TBLR and TBLW need three cycles for execution. Temporarily, the PC value is pushed into the stack and table address is loaded in PC. Therefore care about the stack must be taken while using TBLR and TBLW instructions. TBLR is useful for reading coefficients that have been stored in Program ROM, or time dependent data stored in program RAM. IN and OUT instructions need two machine cycles for execution whereas TBLR and TBLW need three machine cycles for execution.

Control Instructions - The OVM register is directly under program (Fig. 5.1) control. It is set by SOVM instruction and reset by the ROVM instruction. If an over flow occurs when set, the most positive or the most negative representable value of the ALU is loaded into the accumulator. Whether it is most positive or the most negative value is determined by the over flow sign. If an

OV	OVM	INTM	ARP	DP
----	-----	------	-----	----

OV : Accumulator overflow Flag Register

OVM : Overflow mode bit

INTM : Interrupt Mask bit

ARP : Auxiliary Register Pointer

DP : Data Memory Page Pointer

FIG. 5.1 TMS 32010 STATUS REGISTER

over flow occurs when reset, the accumulator remains unmodified. PUSH and POP instruction push the stack from accumulator and POP the stack to accumulator respectively. If there is a stack over flow, the deepest level of stack will be lost. If the stack is overpopped, the value at the bottom of the stack will be copied into higher levels, until it fills the stack.

Figure 5.1 shows the status register bits. "OV" is the accumulator overflow flag. Zero indicates that the accumulator has not overflowed. One indicates that an over flow in accumulator has occurred. The BV (branch on over flow) instruction will clear this bit and cause a branch if it is set. "OVM" is the over flow mode bit. Zero means that overflow mode, described earlier, is enabled. The SOVM instruction loads the OVM with a one and ROVM loads the OVM with ^azero. "INTM" is the interrupt mask bit. Zero means an interrupt is enabled. One~~s~~ means the interrupt is disabled. The EINT instruction loads to INTM bit with ^azero; DINT loads the INTM bit with a one. When an interrupt is executed, the INTM bit is automatically set to one before the interrupt service routine begins. "ARP" is auxiliary register pointer. Zero selects ARO, one selects AR1. The ARP can be changed by MAR or LARP instructions. It can also be changed by instructions that permit indirect addressing option. "DP" is the data^a memory page pointer. Zero selects first 128 words of data memory, i.e., page zero. One selects last 16 words of data memory, i.e., page one. The DP can be changed using LDP or LDPK instructions. The contents of the status register can be stored in data memory by executing the SST instruction. The LST instruction reloads

the status register, with the exception of the INTM bit. The INTM bit cannot be changed through LST instruction.

BIO. INTERRUPT AND RESET - The BIO pin on TMS32010 is an external pin which supports bit test and jump operations. When a low is present on this pin, execution of BIOZ instruction will cause a branch to occur.

The TMS 32010 interrupt is generated either by applying a negative going edge to the interrupt pin or by holding the interrupt pin low. If the interrupt mode register (INTM) is set, then an internal interrupt signal becomes valid. This causes a branch to location 2 in program memory. The interrupt servicing is delayed in each of the following cases:

- 1) Until the end of all cycles of a multicycle instruction.
- 2) Until the instruction following the MPY or MPYK has completed execution.
- 3) Until the instruction following EINT has been executed (when interrupts have been previously disabled). This allows the RET instruction to be executed after interrupts become enabled at the end of an interrupt routine.

The Reset function is enabled when an active low is placed on the R_s pin for a minimum of five clock cycles. The PC is cleared and interrupts are disabled. The over flow mode register (OV) remains unchanged on Reset.

5.2 SIMULATION PROCEDURE

The simulation program treats TTY as the console for the user. The main simulator program which directs the program flow to "CMDPROC" procedures and after completing the desired task comes back to the console mode is reproduced below:

```
begin
    INITBRK; CMDTABLE;
    ELGTR = false; INTRFLG; = false; PREVINTR; = false;
    repeat
        WRITE (TTY, '>'); BREAK (TTY);
        GETLINE; GETSYM;
        if SYM = IDENTIFIER then CMDPROC
        else ERROR
    Until COMMAND = 'EXIT'
end ,
```

"GETLINE" is a procedure which copies the entire line of upto 80 characters entered on the TTY, into an internal array. "GETSYM" procedure analyzes this array by separating symbols. It observes the first nonblank character and starts copying. Subsequent characters into another array of length 8, till a blank character is observed. No valid symbol of the simulator is more than 8 character width. If the symbol contains less than 8 characters, then remainder of the array is filled with blanks. A symbol can be one of the following: Identifier,

Number, Slash, Line end and Undefined. GETSYM procedure returns one of them and in case symbol is a number it also returns, with the internal value of the number (i.e. integer representation, discussed in Section 5.3).

If the symbol is a command identifier, flow of the simulator program is directed to procedure "CMDPROC". Table 5.1 lists all the commands accepted by the simulator. A binary search of command table is made to pinpoint the command. If a valid command is found, then the case statement directs the flow of control ^{to} to serve that command.

Modularity of the program enables to break the problem into smaller tasks. After completing the desired task, the simulator returns to the simulator prompt mode until a logical termination of the simulator is asked by the user by giving EXIT as the command.

"EX" and "GO" commands listed in the Table 5.1 are for the single step or continuous execution of the program ^{stored} ~~graded~~ in the program memory and pointed to by the contents of the program counter. The GO procedure calls the execution procedure repeatedly. The steps taken by the simulator to execute instructions continuously is explained below with the help of the flow-chart shown in Fig. 5.2. It is assumed that the program is already residing in the program memory and the program counter contains the address from where the program begins.

TABLE 5.1SIMULATOR COMMAND TABLE
SIMULATOR COMMAND TABLE

COMMAND	ACTION
BRK	Set break point table
DEPDATA	Deposit in data memory
DEPIPORT	Deposit in I PORTS
DEPPROG	Deposit in Program Memory
DEPREG	Deposit in Registers
DISDATA	Display Data Memory
DISOPORT	Display O PORTS
DISPROG	Display Program Memory
DISREG	Displace ^y Registers
EX	Execute One Instruction
EXIT	Stop Simulator Program
GO	Continue Execution
INTR	Interrupt made active at user defined interval
LOAD	Load Program Memory from a File
LSTBRK	List all the breakpoints
PLOT	Graphic representation of Program Memory
REMBRK	Remove break points
REMTTRACE	Remove the Trace Points
RESET	Reset the Processor
SETTRACE	Set the Trace Points.

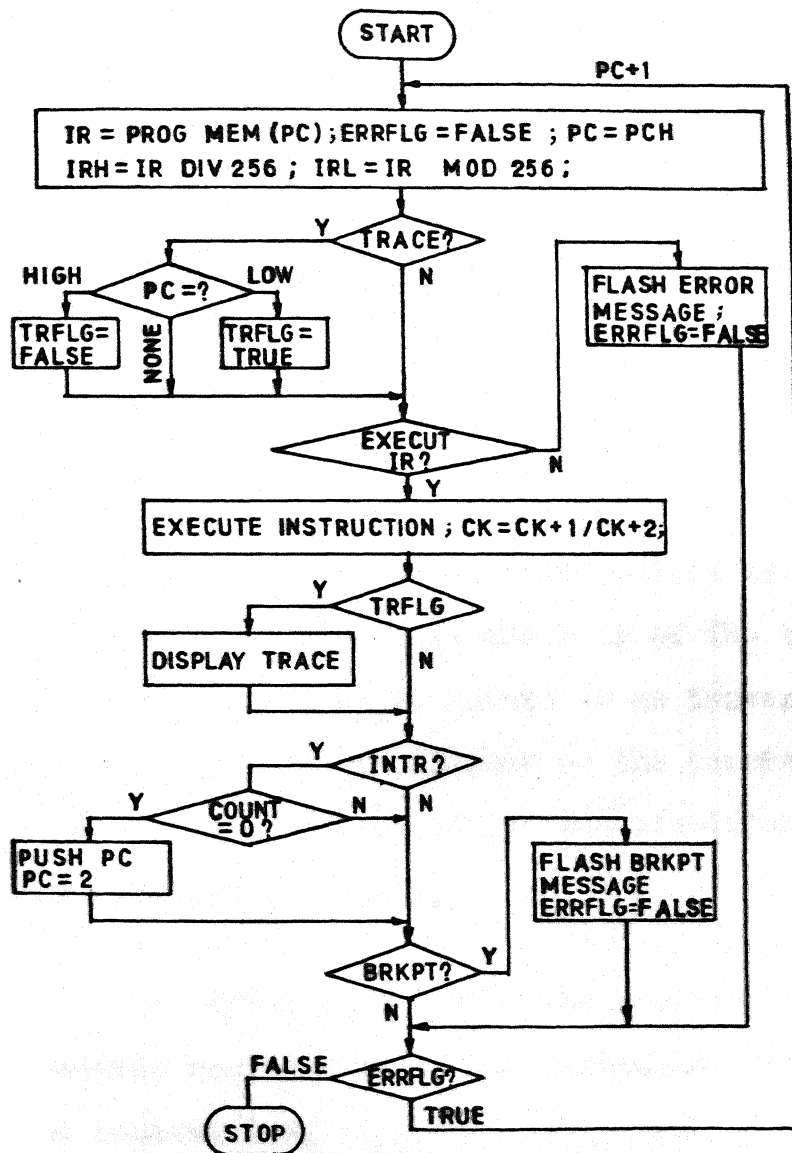


FIG. 5.2 FLOW CHART OF SIMULATION STEPS

1) The contents of the program memory pointed to by the program counter (PC) are loaded into the instruction register (IR). PC is incremented by one. If the trace mode is on, this PC value is compared with low and high values of the trace values setup by the user. If the Trace value equals the low address value then the Trace flag is set true. If the PC value equals the high address value then the Trace flag is made false. If the PC value does not equal either the low address or the high address setup by the user for Trace, then the trace flag remains unaffected. Since TMS 32010 instructions do not have a fixed format, the opcode is not easy to decode. The 16 bit IR is subdivided into IRH and IRL, each of 8 bits. Examination of IRH gives 256 choices, many of which lead to a particular instruction. Cases where the same IRH points to more than one instruction are further analyzed with the help of IRL to pinpoint the instruction. If the IR value points to an inexecutable instruction, then error message is flashed on the terminal, further execution of the program is stopped and the simulator comes back to monitor mode (Simulator prompt).

2) After identifying the instruction, the case statement (decoding routine) calls the particular procedure to carry out that instruction.

3) Each instruction of TMS 32010 is simulated by a separate procedure. The name of the procedure is declared as the mnemonic preceded by P, to give a better readability to the program. After

the control has been passed to the procedure, it finds out the operand by examining IRH or IRL bits. The procedure then fetches the operand and takes proper action. The different registers, memory and status words are changed, clock count is incremented by the required clock cycles and if the instruction needs two words of program memory then PC is further incremented by one.

4) If the Trace flag is on, then the contents of PC, ACC, ARO, AR1, status register and clock counts are displayed on the terminal. The interrupt is made active by the user and the desired number of clock cycles is set by procedure "PINTR". When the desired number of clock cycles have elapsed, then interrupt flag is set true and appropriate action is taken. ^{Actions} to be taken on an interrupt are,

- (1) Reset INTERRUPT flag, Set INTR
- (2) Push PC to stack
- (3) Load PC with 2.

5.3 DESIGN ASPECTS

While designing the TMS 32010 simulation system, following points have been considered:

- (1) The simulation system should truly reflect the architecture of TMS 32010 and should be able to execute the instruction set.

(2) It should have sufficient software peripheral, e.g. input file, output file, load, plot, attached to the processor to carry out tasks encountered in speech processing.

(3) Sufficient debugging features must be provided for efficient and accurate program development.

This section discusses the design of the simulator to meet the above.

5.3.1 General:

A block diagram of TMS 32010 simulation system is shown in Fig. 5.3. Each storage location of TMS 32010 is mapped onto a storage location of DEC 1090 system. The mapping is as follows:

TMS 32010

- Data Memory - an array DMEMORY of 144 words of 16 bit.
- Stack - an array STTSACK of 4 words of 12 bit.
- IR, ARO, AR1, and T registers - a separate memory location of 16 bits each having the names IR, ARO, AR and TR respectively.

- PC - a memory location of 12 bits of name PC
- ACC and P register - a separate memory location of 32 bits each
- BIO, OV, OVM, INIM, ARP and DP - a separate memory location of 1 bit each.

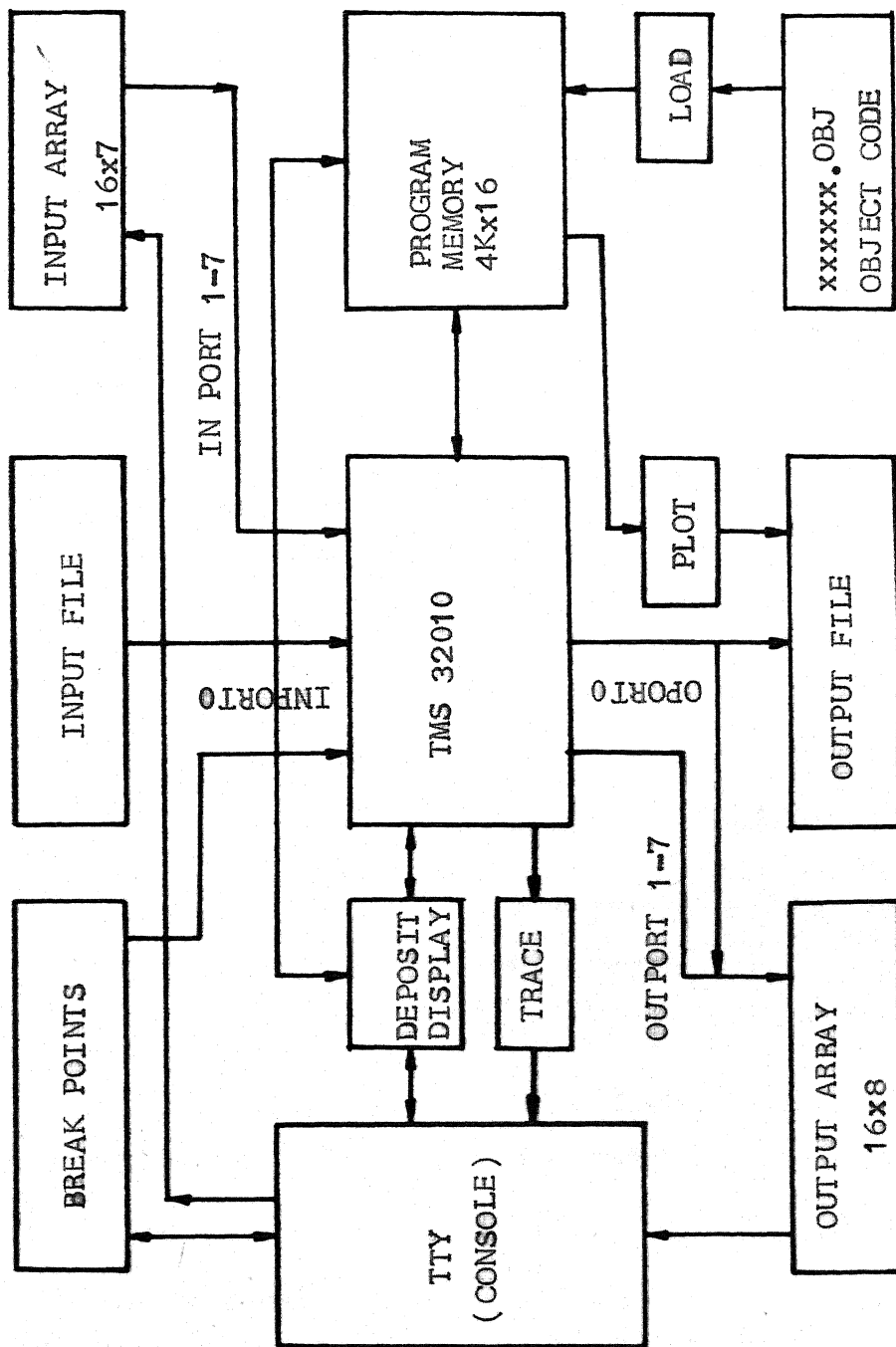


FIG. 5.3 BLOCK DIAGRAM OF TMS 320 SIMULATION SYSTEM

Software Peripherals:

- Program Memory - an array PMEMORY of 4096 words of 16 bits each
- INPORT0 - INPUT file
- INPORT 1 to 7 - an array of 7 words of 16 bit each.
- OUTPORTS - an array of 8 words of 16 bit each.
- OUTPORT 0 - OUTPUT file.

5.3.2 Integer Arithmetic:

TMS 32010 has binary representation of all the words and does ^{2's}~~its~~ complement arithmetic. Set of binary numbers representable in TMS 32010 is mapped on to a set of integers. The rule for representation is a simple binary to decimal conversion rule. Table 5.2 gives the mapping of some typical numbers.

Number	TMS 320	Simulator
Pos Max (32 bit)	> 7FFF FFFF	2**31 - 1
NegMax (32 bit)	> 8000 0000	2**31
+1 (32 bit)	> 0000 0001	1
-1 (32 bit)	> FFFF FFFF	2**32 - 1
0 (32 bit)	> 0000 0000	0
Pos Max (16 bit)	> 7 FFF	2**15 - 1
NegMax (16 bit)	> 8000	2**15
+1 (16 bit)	> 0001	1
-1 (16 bit)	> FFFF	2**16 - 1
0 (16 bit)	> 0000	0

TABLE 5.2 : Mapping of TMS 320 words to Simulator Words.

Operations of arithmetic shift, add, subtract, and multiply when performed using integer arithmetic need intermediate storage. "SCRATCH" and "TEMP" are the variable names given to these locations in the simulator program.

TMS 32010 has 32 bit fixed point arithmetic. $\text{MOD } 2^{**32}$ is the equivalent operation used in the simulator program to truncate the results. A 16 bit data word is sign extended and treated as a 32 bit word by TMS 32010 ALU. For this conversion the simulator checks as follows:

If $\text{number} \geq 2^{**15}$ then $\text{number} = 2^{**32} - 2^{**16} + \text{NUMBER}$.

Two numbers of similar sign when added if produce a result (after mod 2^{**32}) of different sign then over flow is declared. ^{Subtraction} ~~Saturation~~ of two numbers is carried out by finding ^{10's} ~~2's~~ complement of the number to be subtracted and then adding it. For multiplication, of two numbers, absolute value of the numbers and their sign is recorded. After multiplying the absolute values, if the signs of multiplicand and the multiplier don't match then following conversion is made:

$\text{Result} = 2^{**32} - \text{Result}$.

Result of none of the operations takes a negative value, which is a fatal condition.

For logical operations individual bit of a number has to be tested. 'div' and 'mod' operations in PASCAL are useful in identifying I^{th} bit in an integer as follows:

$I^{\text{th}} \text{ bit} = (\text{Number div. } 2^{**I}) \text{ mod } 2$

5.3.3 Interrupts:

Interrupts in speech signal processing are usually generated at regular intervals. The simulator provides the user a facility to generate the interrupts at the preselected clock rate. If the user decides to generate the interrupts, he enters the interrupt rate in terms of clock cycles by using "INTR" command.

Interrupt servicing is delayed until the end of a multicycle instruction in the TMS 32010 processor. In the simulator, each instruction is executed by entering the execute procedure. Hence it is good enough if the clock count is examined after the execution cycle is over. The equation,

$$\text{CLOCK mod RATE} = 0 \quad (5.1)$$

gives the instant when interrupt arrives. Since TMS 32010 has multicycle instructions and interrupt servicing has to be delayed until the end of multicycle instruction, equation 5.1 fails to generate the interrupts at the rate of every "RATE" clock cycles. A solution is to modify equation 5.1 to

$$\text{CLOCK mod RATE} = 0, \text{ or } 1, \text{ or } 2 \quad (5.2)$$

and introduce a flag "PREVINTR". This flag distinguishes between the cases where R.H.S. of equation 5.2 is 0 followed by 1 or 2 and the cases where 0 is skipped and R.H.S. of equation 5.2 is 1 followed by 2. This is sufficient because no instruction of TMS 32010 takes more than three cycles for execution.

Interrupt processing has to be delayed by one instruction in case the current instruction is MPY or MPYK or EINT. A flag "DELAY" is set true when any one of the above instruction is executed. This flag is reset at the beginning of execution of any instruction.

The logic to generate the interrupt is as follows:

```

If CLOCK mod RATE = 0
then (1) INTERRUPT = true
      (2) PREVINTR = true, else
If (clock mod RATE = 1 or 2) and (PREVINTR = false)
then
  (1) INTERRUPT = true
  (2) PREVINTR = true
else PREVINTR = false.

```

When the "INTERRUPT" flag is true, "DELAY" flag is false and INTM = 0 then the simulator goes for interrupt servicing by pushing the PC value in the stack and loading PC by 2.

'INTERRUPT' flag is reset only when interrupt is serviced.

5.3.4 Load:

The object code file generated by the assembler is to be loaded into the program memory. This is achieved by providing a "LOAD" facility in the simulation system (Fig. 5.3). Standard extension to the object code file is OBJ. The instructions for

attaching the user defined file name to the internal file name are similar to that used for assembler and are described in details in Section 4.5. Standard PASCAL provides RESET (F) and REWRITE (F) statement to use external file. Extension to these statements are available in the PASCAL compiler used in DEC system 1090 computer at IIT Kanpur. The statements used in the simulator program to attach the user defined file name to the internal file is RESET (F, File name). Since other fields such as Project, Programmer number and protection code etc. are not used the file is looked into the user's area as a default mode.

5.3.5 I/O Ports:

TMS 32010 provides 8 inports and 8 outports. The simulation system (Fig. 5.3) is designed to provide adequate facility to run application programs without any constraints. For this purpose INPORT 0 is attached to the input file. Since the size of the input file is limited by the memory made available to the user, a good amount of data can be stored. 100 blocks of memory can store ^{2.5} seconds of speech data at 10 KHz sampling rate. INPORT 1 to 7 are provided as 16 bit registers. Provision is made to load the data into the ⁿinport 1 to 7 from the TTY.

The OUTPORT 0 of the simulation system is designed as the OUTPUT file. Data outputted to PORT 0 are written into the OUTPUT file in users area. OUTPORT 0 to 7 are simulated as an array of 8 words, 16 bit each. These ports can be inspected by the user. It may be noted that the data written on OUTPORT 0 appear in the OUTPUT file as well as in the OUTPORT(0) of the array.

5.3.6 PLOT:

In speech processing applications often a plot of the data conveys more meaning than just the numerical values. A need was therefore felt to design the plot facility. The "PLOT" command produces a visual indication of the numeric values of program memory. In TMS 32010 applications, a block of data may reside in the program memory. The processed data is put back in the program memory. A typical application may be FFT, Autocorrelation, Average Magnitude Difference Function (AMDF) of a block of data. Plot procedure is designed as follows:

16-bit number has a range from 0 to 6553⁵₆. Down scaling by a factor of 820 gives the range 0 to 80 which can be represented on a terminal with a width of 80 characters. To plot the graph, "LOW" and "HIGH" limits of the Program Memory address are set by the user. To start with, PC is loaded with the low value and Program Memory contents of the location pointed to by the PC are fetched. This is downscaled as discussed earlier. The down scaled number is decremented until it becomes zero. At each decrement, a blank is written on the output file. When the downscaled number becomes zero a "*" is written on the output file, PC is incremented, and the steps listed above are carried out until the PC reaches the "HIGH" value provided by the user.

The PLOT facility uses the output file to store the graphical representation. Linking of the OUTPORT 0 to the OUTPUT file and use of PLOT command cannot be done simultaneously. This is not a drawback since in usual practice if the user takes data from program memory then he returns the result also to the program memory; (i.e. Program Memory acts as a source of time dependent

data).

5.3.7 TRACE:

During the course of execution, different register contents do change according to the program. For debugging and efficient program development it was thought to provide a facility by which user is made aware of these changes at every instruction. As shown in Fig. 5.3 the TRACE block does this function. A Trace flag "TRFLG" is set when the user opts for Trace facility by using "SETTRACE" command. Low address and High address provided by the user set the begin and end of the Trace. When the PC value during execution equals the Low address, another flag "FLGTR" is set True, and when the PC value during execution equals the HIGH address the flag "FLGTR" is set false. As long as the flag "FLGTR" is ^{true} ~~time~~, trace is displayed. The "TRFLG" flag set true by the SET-TRACE command can be made false by REMTRACE command thus disabling the trace. Registers which are shown on the terminal during trace are - PC, ACC, ARO, AR1, status and number cumulative of clock cycles.

5.3.8 Deposit and Display:

A very useful part of the TMS 320 simulation system of Fig. 5.3 is deposit and display block. It acts as an interface between the user and the processor. Deposit facility has been designed to store user defined values in Program Memory, Data Memory, In Ports 1 to 7, and various registers such as PC, ACC, PR, ARO, AR1, IR and TR. This gives the user enough command to control the processor manually. Display facility

displays data from Program Memory, Data Memory, out Ports 0 to 7, and various registers such as ARO, AR1, TR, IR, PC, ACC and PR. The display facility allows the user to inspect various locations which usually cannot be inspected on a hardware system.

5.3.9 Break Points:

When the program is continuously executed it is some times necessary to temporarily suspend the program execution and use the display/deposit commands to examine the contents of different locations which are usually not available by the Trace, and/or change the contents of different registers. This is done by providing break points to the simulation system of Fig. 5.3.

Break points are represented by an array "BRKPT" of boolean variable. There are 4096 break point flags, each corresponding to one program memory location. Size of the break table is 8 and is arranged as a record. Thus at most 8 different break points can be set. When a break point is set, at a particular program memory location, the entry corresponding to that location in "BRKPT" array is made true. During program execution, execution is suspended if the "BRKPT" flag corresponding to that PC value is true.

Facility is provided to list the break points and remove the break points. Removal of break points is achieved by settling the "BRKPT" array flag corresponding to the desired location as false. Selective insertion and removal of breakpoints helps the user work ~~work~~ efficiently with 8 break points.

5.4 SPECIAL FEATURES

(1) During program development, the concept of modular programming is strictly adhered to. Each instruction is simulated by a separate procedure. Other services, like decoding, effective address calculations, over flow detection etc., are achieved by separate procedures.

(2) Adequate documentation facility is provided in the source program to explain the working of the simulator in general and constituent subroutines in particular.

(3) Particular emphasis is given to ensure that the simulator takes a minimal path in simulating instructions. Use of CASE statement available in PASCAL for instruction decoding help to this end.

(4) Addition of new instructions is not restricted by the simulator. For addition of each new instruction, a little modification in the CASE statement is needed to identify that instruction and another procedure is to be added to execute the instruction.

(5) Total number of clock cycles the program requires for execution is calculated to give the user an idea of actual execution time.

(6) Warning messages and error messages are flashed on the terminal. PC location in question is also given which helps the user in program development.

(7) Reasonable amount of debugging facility is provided to let the user know/modify intermediate results, memory contents, registers etc.

5.5 HOW TO USE THE SIMULATOR

When the simulator is run, a '[>]7' prompt character appears on the terminal. The user can choose any of the commands listed in command table (Table 5.1). The facilities provided by the simulator can be combined in the following groups:

1. Deposit and Display Facilities
2. Trace Facilities
3. Break Point Facilities
4. Execution Facilities
5. Miscellaneous Facilities.

Use of these simulator facilities is explained in details in the following subsections. Most of the conversation between the simulator and the user is interactive and obvious. With a little effort one can know how to use the simulator. In the following discussion CR means carriage return. All numeric values accepted by the simulator are in hexadecimal notation. Message typed by the simulator on the terminal screen is enclosed by " : " character.

5.5.1 Deposit and Display Facilities:

Reset: It is equivalent to the resetting of the TMS 32010 by external command on RS line. Except the program memory and the over flow mode register, it clears all other memory locations assigned to the simulator. It does not change the trace mode or the break points, it is ^{invoked} ~~invoked~~ as follows:

```
' > ' RESET CR
```

-Deposit Data Memory: Data can be deposited in the data memory by this command in the user defined data memory location. Subsequent data memory locations are made available for depositing data. It is invoked and used as follows:

```
' > ' DEPDATA aa CR ; aa = data memory address
' aa / ' d0d0d0d0 CR ; dddd = data value
' aa+1/' d1d1d1d1 CR ; continue depositing subsequent
                           locations
' aa+n/' CR ; come out of deposit data mode.
```

-Display Data Memory: Model session for this command is given below:

```
' > ' DISDATA aa CR ; display data memory from location a
' aa/d0d0d0d0' CR ; display contents of aa
' aa+1/d1d1d1d1' CR ;
                           ; displays contents of subsequent
                           locations
' aa+n/dndndndn'/ CR ; comes out of display data mode.
```


-Deposit Program Memory: Data can be deposited in the program memory. Model session of this command is listed below.

```
'>' DEPPROG   aaa   CR;  aaa = program memory address
'aaa/' d0d0d0d0   CR;  d0d0d0d0 data to be deposited
'aaa+1/' d1d1d1d1   CR; continue depositing subsequent locations.
'aaa+n/'      CR; comesout all of this session.
```

-Display Program Memory: Model session for this command is given below.

```
'>' DISPROG   aaa CR ;  aaa = program memory address
'aaa/d0d0d0d0'   CR ;  starts displaying from aaa
'aaa+/d1d1d1d1   CR ;  continue displaying
'aaa+n/dndndndn'/ CR ;  comes out of this session.
```

-Deposit IN Port: Model session for this command is given below.

```
'>' DEPIPORT a   CR;  a = port address.
'a/ ' d0d0d0d0   CR;  d0d0d0d0 is deposited in IN port a
'a+1/' d1d1d1d1   CR;  continues depositing in subsequent IN ports
'a+n/'      CR;  comes out of this session.
```

-Display OUT Port: Model session for this command is given below:

```
'>' DISOPORT a CR ;  a = port address.
'a/ d0d0d0d0'   CR ;
'a+1/d1d1d1d1' CR ;  display continues
.
.
'a+n/dndndndn'/CR ;  comes out of this session.
```

- **Display Registers:** One of the registers can be displayed at a time. The registers that can be displayed are ARO, AR1, TR, IR, PC, ACC and PR registers. Since these registers are of different length the output displayed is of appropriate length. Model session for this command is given below:

```
'>' DISREG R           ; R is the register abbreviation given ab
'R/D'                  ; D = contents of the register
'>'                    ; Simulator prompt comes back automatical
```

- **Deposit registers:** One of the following registers can be deposited with the appropriate data value. The registers are ARO, AR1, TR, IR, ACC, PR, PC, ARD, DP, BIO. Model session for this command is given below:

```
'>' DEPREG R           ; R is register abbreviation given above
'R/' D      CR         ; D = appropriate data
'>'                    ; Simulator prompt comes back automatical
```

- **Load:** Program can be loaded automatically in the program memory from the user defined file. This file must be present in the user area with an extension OBJ. Usually, assembler discussed in Chapter 4 produces this file. PC value is automatically set by the ORG pseudo-operation present in the first line of the source program. Model session for load is given below:

```
'>' LOAD      CR
'GIVE OBJ FILE NAME: ' FILE NAME CR
'>' ; Simulator prompt comes automatically after loading.
```

5.5.2 Trace Facilities:

Contents of different registers and memory locations cannot be usually observed unless specific action is taken as described earlier. Trace facility enables the user to read the contents of important registers while the program is being executed. It is very helpful in application program development. A display of PC, ACC, ARO, AR1, status bits and clock cycles is provided when trace is set. The simulator provides the following trace facilities.

-Set Trace: Trace can be set between the two PC values. During program execution when the PC value reaches the lower limit of the trace value, the simulator starts displaying the contents of the important registers mentioned earlier. When during program execution the PC value reaches the high limit of the trace the simulator stops displaying the contents of various registers. Model session of settrace is given below.

```
'> SETTRACE CR
```

```
'LOW: ' L CR ; User, provides the lower limit L of the trace on
the TTY.
```

```
'HIGH': ' H CR; User provides the higher limit H of the trace
on the TTY.
```

- Remove trace: The trace set by the SETTRACE command can be removed by this command. Model session using this command is given below:

```
'> REMTRACE CR.
```

5.5.3 Break Point Facilities:

Trace facility explained earlier gives a limited amount of information to the user. The break point facility allows the user to setup upto a maximum of 8 break points, list the break points and remove the break points. Execution of the program is suspended when a break point is met. The user can inspect and/or alter the internal registers and both program and data memories. With the help of the facilities listed in 5.6.1. At the break point the program returns in the simulation mode and simulator prompt character appears on the TTY. Thus all the facilities enumerated in Section 5.6 are at the user's disposal. Break point facilities^s are given below:

- Set break points: A maximum of 8 break points can be set up. Since the break points themselves can be setup or removed when a break point is met this limit on maximum number of break points is sufficient to test a program of any complexity. Model session to set break points is given below.

'>' BRK B1 B2 ... CR, Sets break points at PC locations B₁, B₂

- Remove break points: User can remove all the break points together or can selectively remove the break points as per the following session example.

'>' REMBRK B₁ B₂....CR; Removes break points at B₁, B₂ etc. selectively.

'>' REMBRK CR; Removes all the break points.

- List the break points: By this command the user can list the break points entered in the break point table. Model session is

given below.

```
'>' LSTBRK CR;
```

5.5.4 Execution Facilities:

The simulator provides the option of single stepping or continuous execution. Interrupt facility also is provided in the simulator to register the interrupts at user defined interval. These execution facilities are given below:

- Single Step Execution: The simulator executes one instruction and returns to the simulator prompt; enabling the user to use all the debug facilities. Example for single stepping is given below:

```
'>' EX CR ; Executes one instruction and comes back to
simulator prompt.
```

- Continuous execution: In continuous execution mode the simulator continues to execute the program until a break point is met or an error has occurred during program execution. Example for continuous execution is given below:

```
'>' GO CR;
```

-Execution with interrupts: User is provided with the facility to generate the interrupts at a given rate. Model session to set up the interrupt is given below:

```
'>' INTR CR;
```

'AT CLOCK INTERVALS: ' N CR; N is the number of clock cycles after which interrupt is generated.

5.5.5 Miscellaneous Facilities:

-Exit: One can stop the simulator program by EXIT command. An example is shown below:

```
'>' EXIT CR.
```

-PLOT: This facility enables the user to plot the contents of program memory in the output file in user's area. The program memory can be plotted. Scaling is done in such a way that the maximum positive and minimum negative numbers are represented by about 40 character spaces enabling the whole plot to be seen within 80 lines. A model session of plot command is shown below:

```
'>' PLOT CR
```

```
'>' : 'L CR; user gives the lower limit of program  
memory
```

```
'HIGH' : ' H CR; user gives the higher limit of program  
memory.
```

Model Session on Simulator for FFT Computation is shown.

5.5.6

MODEL SESSION

XECUTE M.REL
 NK: Loading
 NKXCT M execution]

LOAD

VE OBJ FILENAME : FFT8

SETTRACE

W : D7

GH : D9

BRK 108

GO

C:0D8 ACC:0000 0002 AR0:0005 AR1:01FF STATUS: 1 0 0 1 0 CLK: 2681
 C:0D9 ACC:0000 0002 AR0:0005 AR1:01FF STATUS: 1 0 0 1 0 CLK: 2682

DATA ADDRESS OUT OF RANGE AT USER PC:0D9

C:0DA ACC:0000 0002 AR0:0005 AR1:01FF STATUS: 1 0 0 0 0 CLK: 2683
 C:0D8 ACC:0000 0004 AR0:0004 AR1:01FF STATUS: 1 0 0 1 0 CLK: 5047
 C:0D9 ACC:0000 0004 AR0:0004 AR1:01FF STATUS: 1 0 0 1 0 CLK: 5048

DATA ADDRESS OUT OF RANGE AT USER PC:0D9

C:0DA ACC:0000 0004 AR0:0004 AR1:01FF STATUS: 1 0 0 0 0 CLK: 5049
 C:0D8 ACC:0000 0008 AR0:0003 AR1:01FF STATUS: 1 0 0 1 0 CLK: 7261
 C:0D9 ACC:0000 0008 AR0:0003 AR1:01FF STATUS: 1 0 0 1 0 CLK: 7262

DATA ADDRESS OUT OF RANGE AT USER PC:0D9

C:0DA ACC:0000 0008 AR0:0003 AR1:01FF STATUS: 1 0 0 0 0 CLK: 7263
 C:0D8 ACC:0000 0010 AR0:0002 AR1:01FF STATUS: 1 0 0 1 0 CLK: 9399
 C:0D9 ACC:0000 0010 AR0:0002 AR1:01FF STATUS: 1 0 0 1 0 CLK: 9400

DATA ADDRESS OUT OF RANGE AT USER PC:0D9

C:0DA ACC:0000 0010 AR0:0002 AR1:01FF STATUS: 1 0 0 0 0 CLK: 9401
 C:0D8 ACC:0000 0020 AR0:0001 AR1:01FF STATUS: 1 0 0 1 0 CLK:11499
 C:0D9 ACC:0000 0020 AR0:0001 AR1:01FF STATUS: 1 0 0 1 0 CLK:11500

DATA ADDRESS OUT OF RANGE AT USER PC:0D9

C:0DA ACC:0000 0020 AR0:0001 AR1:01FF STATUS: 1 0 0 0 0 CLK:11501
 C:0D8 ACC:0000 0040 AR0:0000 AR1:01FF STATUS: 1 0 0 1 0 CLK:13580
 C:0D9 ACC:0000 0040 AR0:0000 AR1:01FF STATUS: 1 0 0 1 0 CLK:13581

DATA ADDRESS OUT OF RANGE AT USER PC:0D9

C:0DA ACC:0000 0040 AR0:0000 AR1:01FF STATUS: 1 0 0 0 0 CLK:13582

DATA ADDRESS OUT OF RANGE AT USER PC:0DF

Break-at-user-pc: 0108

> PLOT

LOW : 00

HIGH : 7F

> EXIT

EXIT

TV OUTPUT

5.6 CONCLUSION

The simulator is a useful tool to develop and test the program. This simulator is very helpful in developing signal processing programs to be run on TMS 32010. Randomization of interrupt is not done purposely because in speech processing applications the interrupts occur at regular intervals. This is in conformation with the simulator supplied by the manufacturers of TMS 32010. Documentation is provided in the simulator program to understand its working. At present, the output file in users area is used for two purposes. Out put file acts as OUTPORT 0 and also gives the plot of program memory. It is possible to work out with this arrangement. Data is either stored in the program memory and out putted in the program memory, or else data is stored in the Input file and result of processing is outputted in the output file.

REFERENCES

- [1] Texas Instruments, "TMS 32010 User's Guide", Revision A, Nov. 1983.
- [2] Nirmal Roberts et al., "6502 Simulator and Debugger", C.S. Deptt., IIT Kanpur, June, 1986.
- [3] B. Dash, "M 68000 Simulation System", M.Tech. Thesis, C.S. Deptt., IIT Kanpur, Aug. 1983.
- [4] T.V. Sreenivas, "Simulation of a Programmable Signal Processor", Signal Processing 6 (1984) pp. 135-142.

CHAPTER 6

RESULTS

6.1 INTRODUCTION

An extensive testing of the assembler and simulator programs is carried out with each individual instructions. In order to generate more confidence in the system ~~as~~ a few bench mark ~~programs~~ have also been tested. By the bench mark programs correctness of the simulator is checked. Exact times can be obtained by them to judge the real time programs. An antialiasing digital filter [1] and FFT program for 64 point complex data [2] are implemented on TMS 320 simulator. Section 6.2 gives the details of digital filter implementation. Filter specifications, design equation, program flow chart, assembler output listing and the impulse response and step response of filter are given. Section 6.3 gives the FFT program. Program description, flow chart, assembler listing, the FFT of real symmetric 8 point square wave and FFT of a symmetric two imaginary points is given. Section 6.4 gives conclusion.

6.2 DIGITAL FILTER IMPLEMENTATION

The filter specifications and the design equation for the digital filter are taken from [1] and are reproduced below.

6.2.1 Filter Specifications:

Sample frequency (f_s)	10 KHz
Corner frequency (f_c)	2 KHz
Attenuation at f_c	-2 db
Attenuation at $1.2 f_c$	-15 db
Passband ripple	± 1.5 db

6.2.2 Design Equation:

The FIR filter for the specifications of 6.2.1 is a 17 tap digital filter. The difference equation is

$$\begin{aligned} y(n) &= -7547 x(n) + 5109 x(n-1) + 7247 x(n-1) + 3367 x(n-3) \\ &\quad - 3685 x(n-4) - 4868 x(n-5) + 6707 x(n-6) + 24279 x(n-7) \\ &\quad + 32767 x(n-8) + 24279 x(n-9) + 6706 x(n-10) - 4860 x(n-11) \\ &\quad - 3685 x(n+12) + 3667 x(n-13) + 7247 x(n-14) + 5109 x(n-15) \\ &\quad - 7547 x(n-16) \\ &= \sum_{n=1}^{17} C_n * X_n, \quad C_n = \text{Constant for tap } n, \\ &\quad X_n = \text{data value at tap } n. \end{aligned}$$

6.2.3 Flow Chart and Program Details:

The flow diagram for the assembly language program to implement the above filter is given in Fig. 6.1. Constants are defined in the program memory from location 1024 onwards. These constants are read into data memory locations 0 to 16. Using TBLR instruction, LOOP A1 achieves this loading. The input data

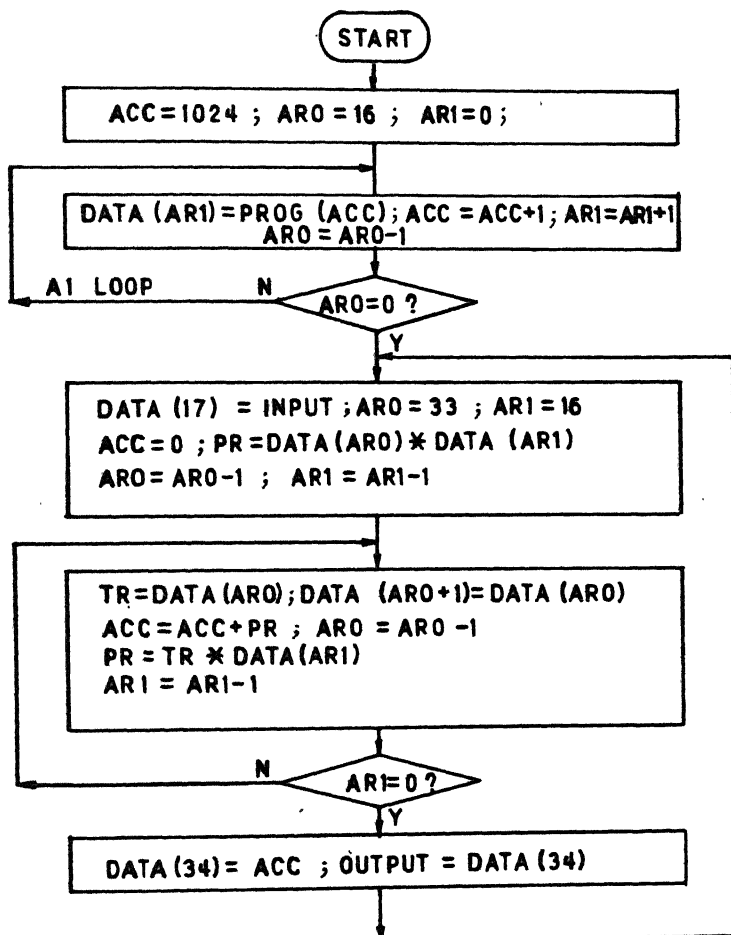
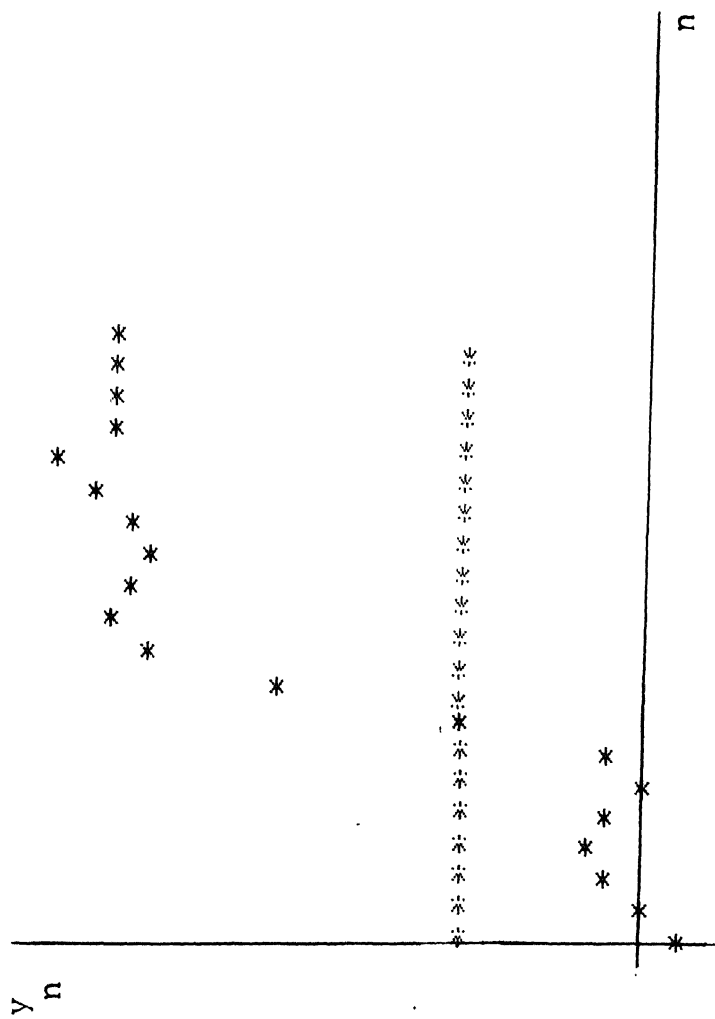


FIG. 6.1 DIGITAL FILTER PROGRAM FLOW CHART



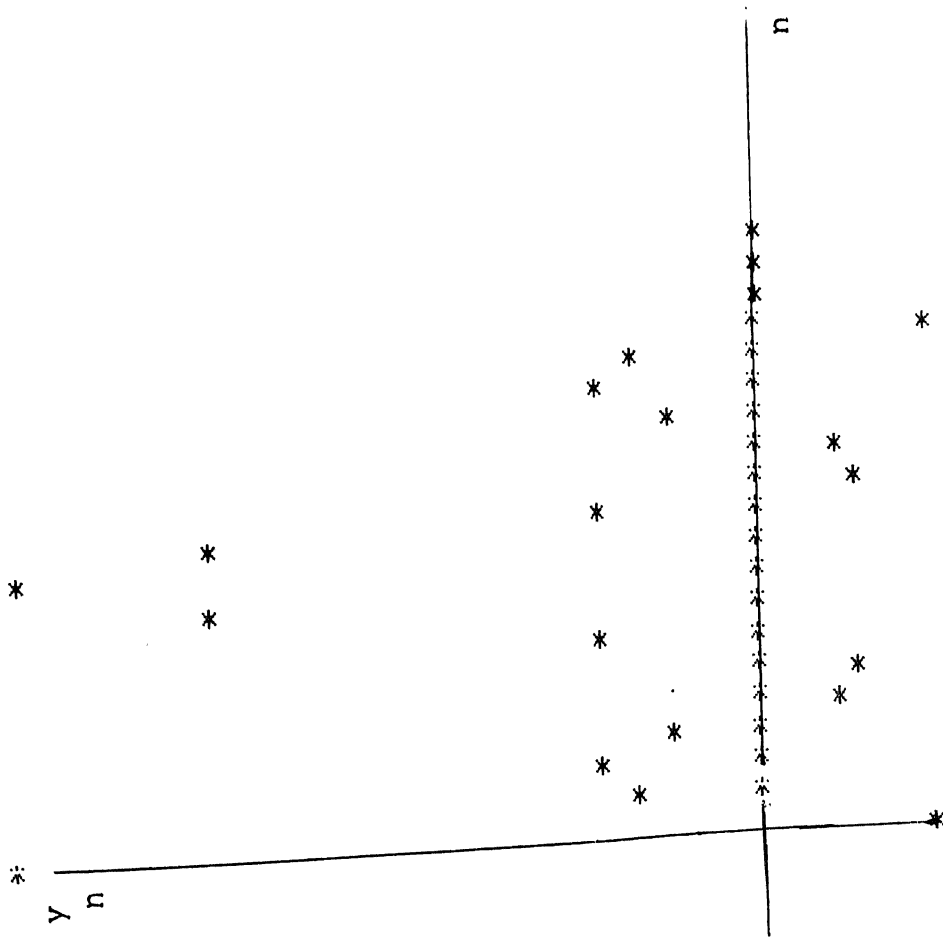


FIG. 6.3 : IMPULSE RESPONSE

080	1		ORG 128		
080	2		LACK 1		
081	3		SACL 127		
082	4		LT 127		#TR=1
083	5		MPYK 1024		
084	6		PAC		#ACC=START ADDRESS FOR
085	7		LARK AR0 16		#AR0 AS LOOP COU
086	8		LARK AR1 0		#
087	9	A1:	LARP 1		#LOAD CONSTANTS.
088	10		TBLR *+ 0		
089	11		ADD 127		#INC ACC BY 1.
08A	12		BANZ A1		
08A					
08C	13	A2:	IN 17 PA0		#INLOOP.
08D	14		LARK AR0 33		#AR0 POINTS DATA ARRAY.
08E	15		LARK AR1 16		#AR1 POINTS CONST AND U
08F	16		ZAC		
090	17		LARP 0		
091	18		LT *- 1		#TR=X17
092	19		MPY *- 0		#C17*X17
093	20	A3:	LARP 0		
094	21		LTD *- 1		#ACCUMULATE AND SHIFT
095	22		MPY *		#MULTIPLY
096	23		BANZ A3		
096					
098	24		APAC		#SUM OF PRODUCTS TO ACC
099	25		ADD 127 14		#ROUND UP.
09A	26		SACH 34 1		
09B	27		OUT 34 PA0		#OUTPUT.
09C	28		B A2		#GOTO IN LOOP.
09C					
400	29		ORG 1024		
400	30		ZEQ - 7545 5109 7247 3667 - 3685 - 4868		
401					
402					
403					
404					
405					
406					
407					
408	31		ZEQ 32767		
409	32		ZEQ 24279 6707 - 4868 - 3685 3667 7247		
40A					
40B					
40C					
40D					
40E					
40F					
410					

ASSEMBLER LISTING FOR FILTER PROGRAM

read into data memory and the filter equation is calculated. By starting constants at location zero in data memory the pointer for constants, AR1 is also used as the loop counter (Loop A2). LTD instruction is used here very effectively. It does the following functions: (a) loads T register, (b) Accumulates previous product, (c) moves data memory, (d) changes ARP. Details of the data memory location used for intermediate computation is given below

- Data (0) to Data (16) : Filter Coefficients C_1 to C_{17}
- Data (17) to Data (33): Input Samples X_1 to X_{17}
- Data (34) : Digital Filter Output.

6.2.4. Results:

Step response of the filter to a step of >2000 is shown in Fig. 6.1. The response saturates to the sum of the taps weighted by the step after 17 samples. The impulse response of the filter is shown in Fig. 6.3. The output of the filter dies down after 17 samples. The number obtained on the output file are recorded in the program memory. Plot facility is then used to get Figs. 6.1 and Fig. 6.3. The program needs 95 clock cycles to output one sample.

6.3 FFT Program:

Colley-Tukey Radix-2 Decimation in frequency FFT program is run on the simulator. The program is for 64 point complex input data. However, it is general enough to accommodate FFT of any size provided. The sin/cos table of suitable length

a block. No scaling is done on the intermediate values in the program. Data are assumed to be stored in the program memory before the program execution. They are stored from location 0. The sin and cos tables are also stored in the program memory. They start from location 1024. The program starts from program memory location 128.

6.3.1 Program Description and Flow Chart:

The signal flow diagram for FFT computations as applied to write the assembly language program is given in Fig. 6.2. Data memory locations on page 0 are used for computations and for storage of intermediate results. There are listed in Table 6.1. Data is arranged in such a way that the real part of the complex input is followed by the imaginary part. Sin and cos values are need for half cycle only. Since there is an overlap between Sin and cos tables, a storage of $3N/4$ values is sufficient to compute the FFT of N points. Loop A3 computes a single butterfly. Butterfly operations are depicted in an inset in the flow chart of Fig. 6.4. The A2 loop updates the theta values (sin and cos functions) whereas the A3 loop is for the nodal computations. The output which appears in the bit reversal order is bit reversed to appear in natural order. The result thus is available in the same program memory locations where the data were stored.

TABLE 6.1 : Data Memory Allocation for FFT Computations

LOCATION	VALUE	COMMENT
0	X(I)	Array value X(I)
1	Y(I)	Array value Y(I)
2	X(L)	Array Value X(L)
3	Y(L)	Array value Y(L)
4	X(I)	Temporary - real part
5	Y(I)	Temporary - imaginary part
6	I	first index
7	L	second index
8	Cos	Twiddle factor - real part
9	Sin	Twiddle factor -imaginary part
10	IA	Index to twiddle factors
11	IE	Increment to IA
12	64	Contains value N
13	I6	Contains value N/4
14	N1	Increment to I
15	N2	Separation of I and L
16	J	Loop counter
17	1	Numerical value 1
18	1024	Coefficient table starting point

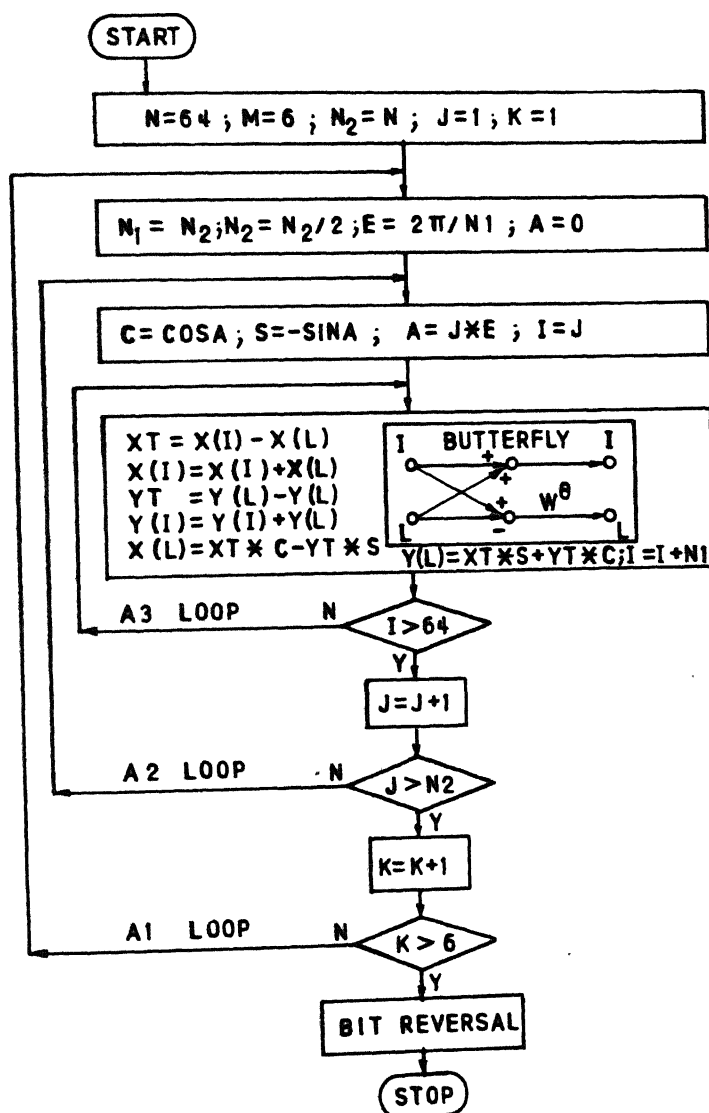


FIG. 6.4 FFT PROGRAM FLOW CHART

FFT PROGRAM

```

) 080      1      ORG 128  ;PROG STARTS.
) 080      2      LDPK 0   ;DP MADE ZERO.
) 081      3      LACK 1   ;ACC=1.
) 082      4      SACL 17  ;DATA(17)=1.
) 083      5      SACL 11  ;DATA(11)=1.
) 084      6      LT 17    ;TR=1.
) 085      7      MPYK 1024
) 086      8      PAC      ;ACC=1.
) 087      9      SACL 18  ;DATA(18)=START OF COS TABLE.
) 088     10      MPYK 64
) 089     11      PAC
) 08A     12      SACL 12  ;DATA(12)=64.
) 08B     13      SACL 15  ;INITIALIZE N2=N.
) 08C     14      LAC 12 14
) 08D     15      SACH 13  ;DATA(13)=N/4.
) 08E     16      LARK ARO 5 ;ARO CONTAINS K COUNTER.
) 08F     17      A1:    LARP 1   ;K LOOP.
) 090     18      LAC 15 15
) 091     19      SACH 14 1 ;N1=N2.
) 092     20      SACH 15  ;N2=N2/2.
) 093     21      ZAC
) 094     22      SACL 10
) 095     23      SACL 16
) 096     24      LAR AR1 15 ;AR1 CONTAINS J VALUE.
) 097     25      MAR *-   ;START AT N2-1.
) 098     26      A2:    LAC 18  ;J LOOP, TABLE IS FULL SIZE.
) 099     27      ADD 10
) 09A     28      TBLR 9   ;GET TWIDDLE FACTORS.
) 09B     29      ADD 13
) 09C     30      TBLR 8
) 09D     31      LAC 10
) 09E     32      ADD 11
) 09F     33      SACL 10  ;INDEX TO TWIDDLE FACTORS.
) 0A0     34      LAC 16 1
) 0A1     35      SACL 6   ;I=J (DATA ORGANIZED AS REAL -
) 0A2     36      A3:    LAC 6   ;FOLLOWED BY IMAGINARY)
) 0A3     37      ADD 15 1 ;L=I+N2
) 0A4     38      SACL 7
) 0A5     39      LAC 6
) 0A6     40      TBLR 0   ;X(I).
) 0A7     41      ADD 17
) 0A8     42      TBLR 1   ;Y(I).
) 0A9     43      LAC 7
) 0AA     44      TBLR 2   ;X(L).
) 0AB     45      ADD 17
) 0AC     46      TBLR 3   ;Y(L).
) 0AD     47      LAC 0    ;COMPUTE BUTTERFLY
) 0AE     48      SUB 2
) 0AF     49      SACL 4   ;XT=X(I)-X(L).
) 0B0     50      ADD 2 1
) 0B1     51      SACL 0   ;X(I)=X(I)+X(L).
) 0B2     52      LAC 1

```

6700	0E7	101		TBLR	0	
0011	0E8	102		ADD	17	
6701	0E9	103		TBLR	1	
2007	0EA	104		LAC	7	
6702	0EB	105		TBLR	2	
0011	0EC	106		ADD	17	
6703	0ED	107		TBLR	3	
2007	0EE	108		LAC	7	
7D00	0EF	109		TBLW	0	
0011	0F0	110		ADD	17	
7D01	0F1	111		TBLW	1	
2006	0F2	112		LAC	6	
7D02	0F3	113		TBLW	2	
0011	0F4	114		ADD	17	
7D03	0F5	115		TBLW	3	
200C	0F6	116	A6:	LAC	12	;NO SWAP.
5010	0F7	117		SACL	16	;J=N.
2007	0F8	118	A7:	LAC	7	;IN LOOP.
1010	0F9	119		SUB	16	;IF L>=J THEN
FA00	0FA	120		BLZ	A8	
0101	0FA					
5007	0FC	121		SACL	7	;L=L-J
2F10	0FD	122		LAC	16 15	
5810	0FE	123		SACH	16	;J=J/2.
F900	0FF	124		B	A7	
00FB	0FF					
0110	101	125	A8:	ADD	16 1	
5007	102	126		SACL	7	;L=L+J.
2006	103	127		LAC	6	
0111	104	128		ADD	17 1	
5006	105	129		SACL	6	;INCREMENT I
F400	106	130		BANZ	A5	
00E3	106					
7F80	108	131		NOP		;SET BRKPT.
7F80	109	132		NOP		
*400	400	133		ORG	1024	
0000	400	134		ZEQ	0	
0C8B	401	135		ZEQ	3211	
18F8	402	136		ZEQ	6392	
2527	403	137		ZEQ	9511	
30FB	404	138		ZEQ	12539	
3C56	405	139		ZEQ	15446	
471C	406	140		ZEQ	18204	
5133	407	141		ZEQ	20787	
5A81	408	142		ZEQ	23169	
62F1	409	143		ZEQ	25329	
6A6C	40A	144		ZEQ	27244	
70E1	40B	145		ZEQ	28897	
7640	40C	146		ZEQ	30272	
7A7C	40D	147		ZEQ	31356	
7D89	40E	148		ZEQ	32137	
7F61	40F	149		ZEQ	32609	

7FFF	410	150	ZEQ	32767
7F61	411	151	ZEQ	32609
7D89	412	152	ZEQ	32137
7A7C	413	153	ZEQ	31356
7640	414	154	ZEQ	30272
70E1	415	155	ZEQ	28897
6A6C	416	156	ZEQ	27244
62F1	417	157	ZEQ	25329
5A81	418	158	ZEQ	23169
5133	419	159	ZEQ	20787
471C	41A	160	ZEQ	18204
3C56	41B	161	ZEQ	15446
30FB	41C	162	ZEQ	12539
2527	41D	163	ZEQ	9511
18F8	41E	164	ZEQ	6392
0C8B	41F	165	ZEQ	3211
0000	420	166	ZEQ	0
F375	421	167	ZEQ	- 3211
E708	422	168	ZEQ	- 6392
DAD9	423	169	ZEQ	- 9511
CF05	424	170	ZEQ	- 12539
C3AA	425	171	ZEQ	- 15446
B8E4	426	172	ZEQ	- 18204
AECD	427	173	ZEQ	- 20787
A57F	428	174	ZEQ	- 23169
9D0F	429	175	ZEQ	- 25329
9594	42A	176	ZEQ	- 27244
8F1F	42B	177	ZEQ	- 28897
89C0	42C	178	ZEQ	- 30272
8584	42D	179	ZEQ	- 31356
8277	42E	180	ZEQ	- 32137
809F	42F	181	ZEQ	- 32609
*000	000	182	ORG	0
1000	000	183	ZEQ	4096
0000	001	184	ZEQ	0
1000	002	185	ZEQ	4096
0000	003	186	ZEQ	0
1000	004	187	ZEQ	4096
0000	005	188	ZEQ	0
1000	006	189	ZEQ	4096
*078	078	190	ORG	120
0000	078	191	ZEQ	0
0000	079	192	ZEQ	0
1000	07A	193	ZEQ	4096
0000	07B	194	ZEQ	0
1000	07C	195	ZEQ	4096
0000	07D	196	ZEQ	0
1000	07E	197	ZEQ	4096

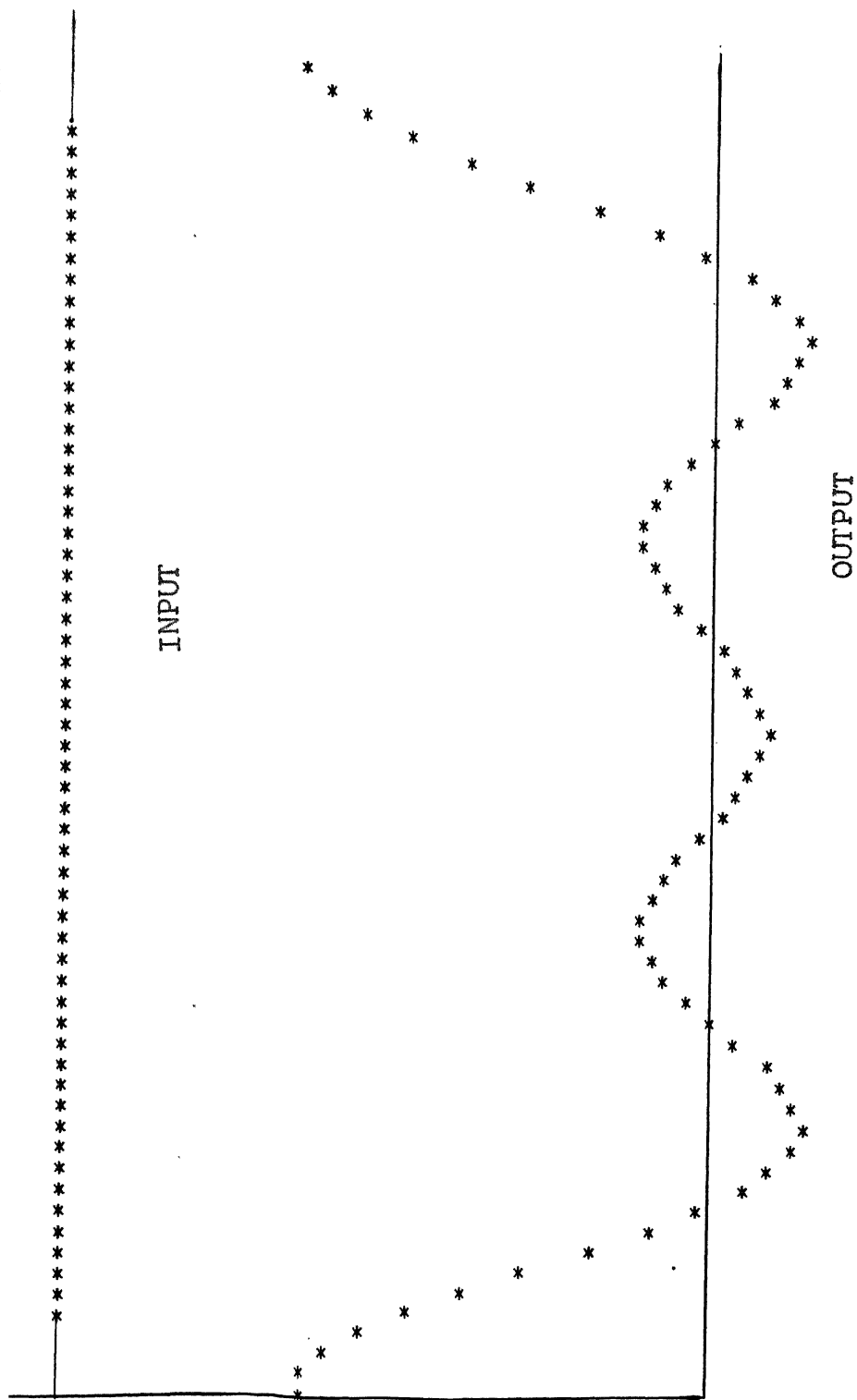


FIG. 6.5 FFT OF REAL SYMMETRIC SQUARE WAVE

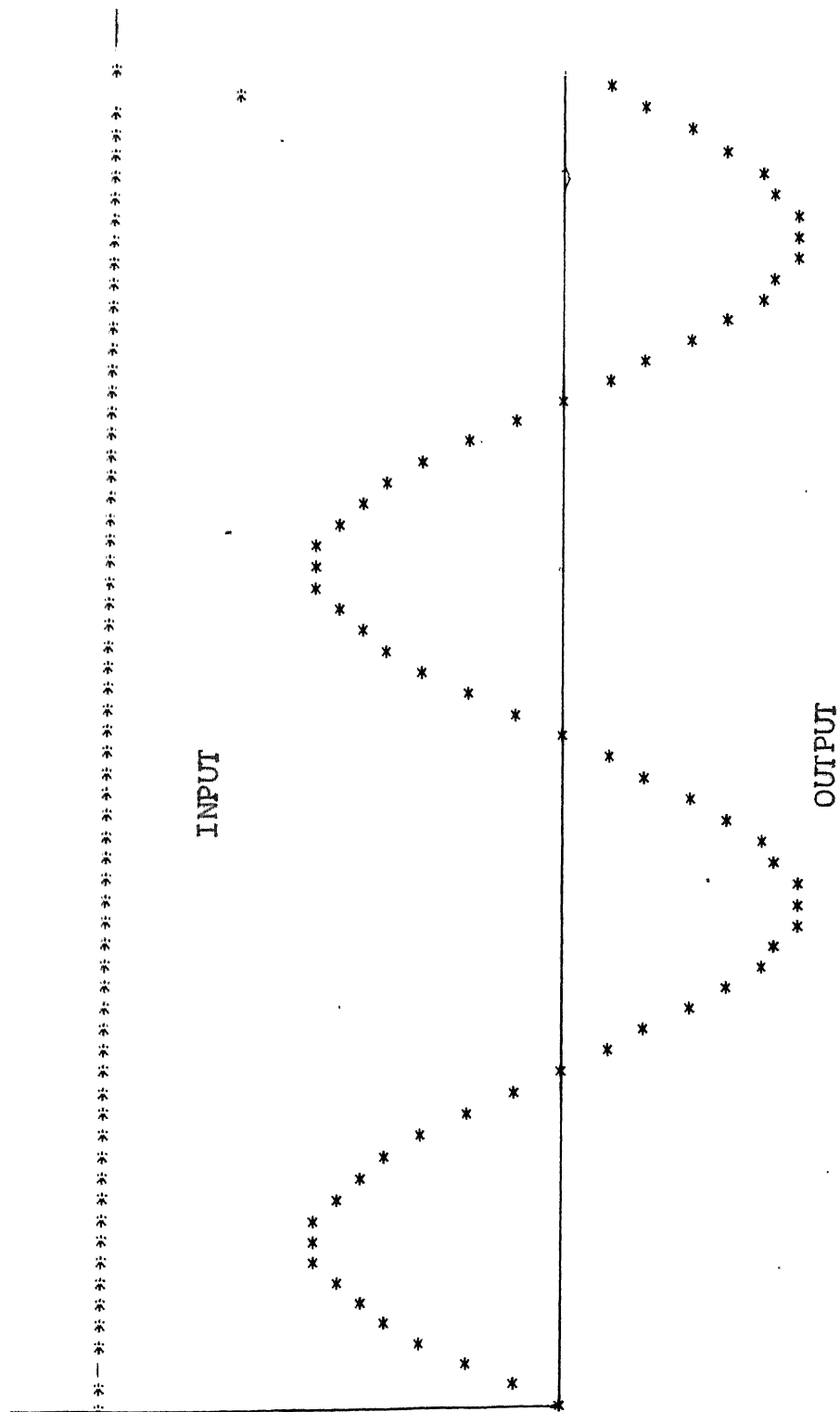


FIG. 6.6 FFT OF IMAGINARY ASYMMETRIC POINTS

6.3² Results:

FFT program was tested by finding the spectrum of a real square wave. The input wave and the output wave are shown in Fig. 6.5. They have been obtained using the plot facility. The spectrum is of $\sin x/x$ shape. Imaginary part of the output which is zero is not plotted. Figure 6.6 shows the FFT output when is put at imaginary location 2 and is put at imaginary location (64-2). The resulting FFT output is a sin wave with 2 cycles. It takes about 16000 (16008) clock cycles to compute 64 point complex FFT.

6.4 CONCLUSION

The bench mark programs run on the TMS 32010 simulation system give the expected results. It shows that the system works perfectly and the facilities provided are adequate. Input/output files attaches to the IPORT and OPORT are used for the digital filter program and the plot facility provided is used for the FFT program. Use of program memory for storing constants is a useful feature of TMS 32010 architecture.

REFERENCES

- [1] "TMS 32010 user's guide", Texas Instruments, Revision A, Nov. 1983, pp. 4-4 to pp. 4-15.
- [2] C.S. Burrus et al., "DFT/FFT and Convolution Algorithms", John Wiley and Sons, New York, 1985, pp. 156-160.

CHAPTER 7

CONCLUSION

7.1 INTRODUCTION

Requirements of a real time speech processor and the architectures based on bit-slice microprocessors are brought out in this thesis. Simulation of Texas instrument's signal processing chip, which is widely used in speech processing has been carried out on the DEC system 1090 computer of I.I.T. Kanpur.

In Section 7.2 conclusions drawn on the speech processor architectures and the advantages of the simulator are presented. A few suggestions are made in Section 7.3 regarding possible future work in the area of speech processor architectures and the type of programs that can be run on TMS 32010 simulator.

7.2 CONCLUSIONS

The following conclusions can be drawn about the speech processor architecture:

- (a) A separate multiplier or multiplier accumulator is needed to multiply or multiply and add in a single machine cycle.
- (b) Speed of the processor must be above 4 Million Instructions Per Second (MIPS).
- (c) The processor must have separate program memory and data memory to enhance the speed.

- (d) It must have powerful branch instructions.
- (e) A stack atleast upto six deep is felt sufficient for speech processing. However an ^{twelve}~~eight~~ deep stack seems to be ideal.
- (f) The I/O devices such as Analog to digital converters, Digital to analog converters, Serial to parallel converters and parallel to serial converters must have direct memory address.
- (g) It should have flexible addressing modes.
- (h) The instruction set must be very powerful and must have special instructions to move data in memory.
- (i) Must have a provision to serve multiple interrupts on a priority basis.
- (j) The microcode must be of reasonable width. 40 to 60 bit microcodes are manageable.

The facilities provided by the simulation system are tailored to the needs of speech processing. However, they can be made use of in other fields such as communications, control seismic processing where the amount of computations are more. Apart from the basic task of simulating the instruction set of TMS 32010 and providing an adequate debugger, the simulator provides the following facilities:

- (a) Input file is attached to the port 0 of the INPORT.
- (b) Output file is attached to the port 0 of the OUTPORT.
- (c) Interrupt generation can be set up at regular intervals

- (d) Graphical representation of the signal can be observed on the terminal screen. A hard copy of the same can also be taken on the line printer.

7.3 SUGGESTIONS FOR FUTURE WORK

The following work can be carried out as a follow up of the work presented in the thesis.

- (1) It would be worthwhile to try the detailed design of an ideal processor, borrowing the good points of the architectures discussed in this thesis. Such a processor when implemented using VLSI technology would be the best answer to the speech processing community.
- (2) A comparative study of different signal processing chips such as Intel's 2920, AMI's 2811, NEC's 7720 and Texas Instruments TMS 320 would be interesting. The present work could not include them due to nonavailability of technical details.
- (3) Digitized speech could be stored in the input file. 200 blocks of memory can store more than 5 seconds of speech. This is sufficient to try out different speech processing algorithms. If speech Analysis-Synthesis techniques are tried out then the output can be played back in real time to see the speech output of that analysis-synthesis technique.
- (4) One of the reasons chosen to use a higher level language for programming was its portability to different computers. PASCAL statements used in the assembler and simulator programs are mostly from Standard PASCAL. The places where it is specific

to the DEC system 1090 are:

- (a) the extension of "case" statement : others
- (b) the extension of "Reset" and "Rewrite" statements.

RESET (F, filename) and REWRITE (F, Filename).

The same program with little modifications to the above two statements must run on other computers such as IBM-PC, PDP-11 and VAX. Although all these machines are 16 bit, the PASCAL implementation on them must be supporting 32 bit integer arithmetic. If 32 bit integer arithmetic is not allowed by the PASCAL compiler on a particular machine then there are major changes in the arithmetic instructions of the simulator.

(5) Generally interrupt at regular intervals is sufficient in speech processing. In the present case the interrupts start with the immediate occurrence of

$$\text{CLOCK Mod RATE} = 0, 1 \text{ or } 2 \quad (7.1)$$

However the interrupt generation could be randomized by doing suitable changes in the software.

(6) The assembler in the present case is an elementary assembler. It can be made more powerful to include MACRO instructions and to allow symbolic names to all the operands.

APPENDIX A

ASSEMBLER SYNTAX

MNEMONIC	ADDRESS	SYNTAX
ABS		[<label>] ABS
ADD	direct	[<label>] ADD <dma>
ADD	indirect	[<label>] ADD {*/**/*-} [<shift> [<ARP>]]
ADDH	direct	[<label>] ADDH <dma>
ADDH	indirect	[<label>] ADDH {*/**/*-} [<ARP>]
ADDS	direct	[<label>] ADDS <dma>
ADDS	indirect	[<label>] ADDS {*/**/*-} [<ARP>]
AND	direct	[<label>] AND <dma>
AND	indirect	[<label>] AND {*/**/*-} [<ARP>]
APAC		[<label>] APAC
B		[<label>] B <pma>
BANZ		[<label>] BANZ <pma>
BGEZ		[<label>] BGEZ <pma>
BGZ		[<label>] BGZ <pma>
BIOZ		[<label>] BIOZ <pma>
BLEZ		[<label>] BLEZ <pma>
BLZ		[<label>] BLZ <pma>
BNZ		[<label>] BNZ <pma>
BV		[<label>] BV <pma>
BZ		[<label>] BZ <pma>
CALA		[<label>] CALA
CALL		[<label>] CALL <pma>
DINT		[<label>] DINT
DMOV	direct	[<label>] DMOV <dma>
DMOV	indirect	[<label>] DMOV {*/**/*-} [<ARP>]
EINT		[<label>] EINT
IN	direct	[<label>] IN <dma> <PA>
IN	indirect	[<label>] IN {*/**/*-} <PA> [<ARP>]
LAC	direct	[<label>] LAC <dma> [<shift>]
LAC	indirect	[<label>] LAC {*/**/*-} [<shift> [<ARP>]]
LACK		[<label>] LACK <constant>
LAR	direct	[<label>] LAR <PA> <dma>
LAR	indirect	[<label>] LAR <PA> {*/**/*-} [<ARP>]
LARK		[<label>] LARK <AR> <constant>
LARP		[<label>] LARP <constant>
LDP	direct	[<label>] LDP <dma>
LDP	indirect	[<label>] LDP {*/**/*-} [<ARP>]
LDPK		[<label>] LDPK <constant>
LST	direct	[<label>] LST <dma>
LST	indirect	[<label>] LST {*/**/*-} [<ARP>]
LT	direct	[<label>] LT <dma>
LT	indirect	[<label>] LT {*/**/*-} [<ARP>]
LTA	direct	[<label>] LTA <dma>
LTA	indirect	[<label>] LTA {*/**/*-} [<ARP>]
LTD	direct	[<label>] LTD <dma>
LTD	indirect	[<label>] LTD {*/**/*-} [<ARP>]

MAR		[<label>]	MAR	{*/+/*-} [<ARP>]
MPY	direct	[<label>]	MPY	<dma>
MPY	indirect	[<label>]	MPY	{*/+/*-} [<ARP>]
MPYK		[<label>]	MPYK	<constant>
NOP		[<label>]	NOP	
OR	direct	[<label>]	OR	<dma>
OR	indirect	[<label>]	OR	{*/+/*-} [<ARP>]
ORG			ORG	<dma>
OUT	direct	[<label>]	OUT	<dma> <PA>
OUT	indirect	[<label>]	OUT	{*/+/*-} <PA> [<ARP>]
PAC		[<label>]	PAC	
POP		[<label>]	POP	
PUSH		[<label>]	PUSH	
RET		[<label>]	RET	
ROVM		[<label>]	ROVM	
SACH	direct	[<label>]	SACH	<dma> [<shift>]
SACH	indirect	[<label>]	SACH	{*/+/*-} [<shift> [<ARP>]]
SACL	direct	[<label>]	SACL	<dma> [<shift>]
SACL	indirect	[<label>]	SACL	{*/+/*-} [<shift> [<ARP>]]
SAR	direct	[<label>]	SAR	<AR> <dma>
SAR	indirect	[<label>]	SAR	<AR> {*/+/*-} [<ARP>]
SOVM		[<label>]	SOVM	
SPAC		[<label>]	SPAC	
SST	direct	[<label>]	SST	<dma>
SST	indirect	[<label>]	SST	{*/+/*-} [<ARP>]
SUB	direct	[<label>]	SUB	<dma> [<shift>]
SUB	indirect	[<label>]	SUB	{*/+/*-} [<shift> [<ARP>]]
SUBC	direct	[<label>]	SUBC	<dma>
SUBC	indirect	[<label>]	SUBC	{*/+/*-} [<ARP>]
SUBH	direct	[<label>]	SUBH	<dma>
SUBH	indirect	[<label>]	SUBH	{*/+/*-} [<ARP>]
SUBS	direct	[<label>]	SUBS	<dma>
SUBS	indirect	[<label>]	SUBS	{*/+/*-} [<ARP>]
TBLR	direct	[<label>]	TBLR	<dma>
TBLR	indirect	[<label>]	TBLR	{*/+/*-} [<ARP>]
TBLW	direct	[<label>]	TBLW	<dma>
TBLW	indirect	[<label>]	TBLW	{*/+/*-} [<ARP>]
XOR	direct	[<label>]	XOR	<dma>
XOR	indirect	[<label>]	XOR	{*/+/*-} [<ARP>]
ZAC		[<label>]	ZAC	
ZALH	direct	[<label>]	ZALH	<dma>
ZALH	indirect	[<label>]	ZALH	{*/+/*-} [<ARP>]
ZALS	direct	[<label>]	ZALS	<dma>
ZALS	indirect	[<label>]	ZALS	{*/+/*-} [<ARP>]
ZEQ			ZEQ	<constant>

Note:

1. Meaning of special symbols is given after the table of contents.
2. First character of a label is an alphabet and second character is a number. Label is always terminated by a semicolon.
3. Since the comment field which concludes the syntax is optional it is not included in the above table. The comment always starts by a colon.

APPENDIX BERROR MESSAGES AND WARNINGS

-Assembler Error messages: Assembler Error code is placed in place of the object code in PAS1 file. It also appears on the Terminal.

Error Code	Description
ERR 1	Illegal character
ERR 2	Label table overflow
ERR 3	invalid label
ERR 4	duplicate label
ERR 5	syntax error
ERR 6	Illegal mnemonic
ERR 7	pma illegal
ERR 8	IIndirect addressing error
ERR 9	DMA error
ERR A	Shift error
ERR B	ARP error
ERR C	Constant out of range
ERR D	PA error
ERR E	AR error
ERR F	pma not found in Pass Two.

User can look at the indicated line number in the source program, correct that error and rerun the assembler. Pass two is executed only when there is no error in Pass One.

- Simulator Error Messages

- a) 'Command Syntax Error' : This error indicates the command to the simulator is not given in the proper format.
- b) 'UNIMPLEMENTED INSTRUCTION AT USER PC : XXX': This message indicates that either the code entered is wrong or some variable during execution has gone out of bounds.

-Warnings:

The array registers are used for indirect addressing. These registers when used as loop counters usually become more than the memory locations (0 to 143). If it is so, a warning message is flashed on terminal as below:

'DATA ADDRESS OUT OF RANGE AT USER PC : XXX '

The user is advised to see the instruction in question and ascertain that he is not using the auxiliary register for addressing the memory at that instant.

PASS1 LISTING

```

*010  010      1      ORG 16 ;TEST PROGRAM.
7F88  010      2      ABS ;
ERR1***      3 A1: 8    ADD 32 10 ;ERROR 1
ERRA***      4      ADD * 18 0 ;ERROR A
ERR4***      5 A1:    ADDH 32 ;ERROR 4
60A1  011      6      ADDH *+ 1 ;
6110  012      7 A2:    ADDS 16 ;
7F8F  013      8      APAC ;
F900  014      9      B A4 ;
*A4    014
F400  016     10      BANC A2 ;
0012  016
FD00  018     11      BGEZ A1;
0011  018
FC00  01A     12 A3:    BGZ A2
0012  01A
F600  01C     13      BIDZ A5 ;FORWARD REFERENCE
*A5    01C
FB00  01E     14 A4:    BLEZ A7 ;FORWARD REFERENCE
*A7    01E
FA00  020     15      BLZ 3A
ERR7***     16      BLZ 3A
FE00  020     17      BNZ A6 ;FORWARD REFERENCE
*A6    020
F500  022     18 A5:    BV A4
001E  022
FF00  024     19      BZ A8 ;FORWARD REFERENCE
*A8    024
7F8C  026     20      CALA
F800  027     21      CALL A6
*A6    027
ERR5***     22      DINT 25 ;ERROR 5
ERR9***     23      DMOV 150 ;ERROR 9
ERR8***     24 A6:    DMOV ** 1 ;ERROR 8
7F82  029     25      EINT
ERRD***     26      IN 16 7 ;ERROR D
4788  02A     27      IN * PA7
2519  02B     28 A7:    LAC 25 5
2CA0  02C     29      LAC *+ 12 0
7E70  02D     30      LACK 112
ERRE***     31      LAR AR3 11 ;ERROR E
7170  02E     32 A8:    LARK AR1 112
70FF  02F     33      LARK AR0 255
6881  030     34      LARP 1
6F10  031     35      LDP 16
6E00  032     36      LDPK 0
7B0C  033     37      LST 12
6AA0  034     38      LT *+ 0
6898  035     39      MAR *-
6880  036     40      MAR * 0
6D40  037     41      MPY 64
6DA1  038     42      MPY *+ 1

```

ASSEMBLER LISTING

```

PROGRAM ASSEMBLER(FINP,FOUP,PAS1);
  label
    100 {USED FOR ERROR TRAP IN PASS 1};
    const
      KING = 65536 {16 BIT WORD};
  type
    WRD =0..KING;
    CMDARRAY = packed array [1..5] of char {FOR MNEMONICS};
    Y = record
      CMD : CMDARRAY
    end {USED IN MNEMONIC SEARCH};
  var
    X :char;
    CC :integer {CHARACTER COUNT };
    LL : integer {LINE LENGTH};
    LINE :array [1..81] of char;
    I : integer {NUMBER OF SYMBOLS IN SYMBOL TABLE};
    J : integer;
    M : integer {LINE NUMBER IN SOURCE FILE};
    B : integer {GETNUM DEPOSITS NUMBER IN B};
    JMPADR :WRD {JUMP ADDRESS FOR BRANCH INST.};
    C:packed array [1..4] of char {OBJECT OR ERROR CODE};
    A : packed array[1..5] of char {GETSYM RETURNS THIS};
    COMMAND :CMDARRAY;
    TABLE : array [1..62] of Y;
    SYM :(SYMBOL,LENEND,INDIRECT,UNDEFINED);
    LC :0..4095 {LOCATION COUNTER};
    INDRCT :boolean {INDIRECT ADDRESS FLAG};
    INTABLE :boolean {LABEL PRESENT IN LABEL TABLE};
    OK : boolean {NO EREROR FLAG ,PASS 1};
    LTBL :array [1..20,1..2] of char {LABEL TABLE};
    VALUE : array [1..20] of WRD {VALUE OF THE LABEL};
    PAS1,FINP,FOUP : TEXT;
    NINP,NOUP : packed array [1..9] of char;
  function INTOHEX(N:integer): char;
  var
    T : char;
  begin
    case N of
      0 : T:='0';
      1 : T:='1';
      2 : T:='2';
      3 : T:='3';
      4 : T:='4';
      5 : T:='5';
      6 : T:='6';
      7 : T:='7';

```

```

PROGRAM ASSEMBLER(FINP,FOUP,PAS1);
  label
    100 {USED FOR ERROR TRAP IN PASS 1};
    const
      KING = 65536 {16 BIT WORD};
  type
    WRD =0..KING;
    CMDARRAY = packed array [1..5] of char {FOR MNEMONICS};
    Y = record
      CMD : CMDARRAY
    end {USED IN MNEMONIC SEARCH};
  var
    X :char;
    CC :integer {CHARACTER COUNT };
    LL : integer {LINE LENGTH};
    LINE :array [1..81] of char;
    I : integer {NUMBER OF SYMBOLS IN SYMBOL TABLE};
    J : integer;
    M : integer {LINE NUMBER IN SOURCE FILE};
    B : integer {GETNUM DEPOSITS NUMBER IN B};
    JMPADR :WRD {JUMP ADDRESS FOR BRANCH INST.};
    C:packed array [1..4] of char {OBJECT OR ERROR CODE};
    A : packed array[1..5] of char {GETSYM RETURNS THIS};
    COMMAND :CMDARRAY;
    TABLE : array [1..62] of Y;
    SYM : (SYMBOL, LNEND, INDIRECT, UNDEFINED);
    LC :0..4095 {LOCATION COUNTER};
    INDRCT :boolean {INDIRECT ADDRESS FLAG};
    INTABLE :boolean {LABEL PRESENT IN LABEL TABLE};
    OK : boolean {NO ERROR FLAG ,PASS 1};
    LTBL :array [1..20,1..2] of char {LABEL TABLE};
    VALUE : array [1..20] of WRD {VALUE OF THE LABEL};
    PAS1,FINP,FOUP : TEXT;
    NINP,NOUP : packed array [1..9] of char;
  function INTTOHEX(N:integer): char;
    var
      T : char;
    begin
      case N of
        0 : T:='0';
        1 : T:='1';
        2 : T:='2';
        3 : T:='3';
        4 : T:='4';
        5 : T:='5';
        6 : T:='6';
        7 : T:='7';
        8 : T:='8';
        9 : T:='9';
        10 : T:='A';
        11 : T:='B';
        12 : T:='C';

```

```

        13 : T:='D';
        14 : T:='E';
        15 : T:='F'
    end;
    INTTOHEX:=T
end { internal-to-hex };
function HEXTOINT(H:char): integer;
var
    T : integer;
begin
    case H of
        '0' : T:=0;
        '1' : T:=1;
        '2' : T:=2;
        '3' : T:=3;
        '4' : T:=4;
        '5' : T:=5;
        '6' : T:=6;
        '7' : T:=7;
        '8' : T:=8;
        '9' : T:=9;
        'A' : T:=10;
        'B' : T:=11;
        'C' : T:=12;
        'D' : T:=13;
        'E' : T:=14;
        'F' : T:=15
    end;
    HEXTOINT:=T
end { hex-to-internal };
procedure ERROR(ABORTS CURRENT LINE PROC,RESETS OK FLAG);
var
    K : integer;
begin
    OK:=false; WRITELN(TTY);
    WRITE(PAS1,C,'***',M:6,' ');
    WRITE(TTY,C,'***',M:6,' ');
    for K:= 1 to LL do
        begin
            WRITE(PAS1,LINE[K]);WRITE(TTY,LINE[K])
        end;
        WRITELN(PAS1); WRITELN(TTY);M:=M+1;
        goto 100
    end;
procedure SYNTAX;
begin
    C:='ERR5'; ERROR
end;
procedure GETLINE {COPIES A LINE FROM SOURCE PROGRAM};
begin
    CC:=0; LL:=0;
    while not EOLN(FINP)do

```

```

        begin
            LL:=LL+1;READ(FINP,X); LINE[LL]:=X
        end
    end {END OF GETLINE};
Procedure GETSYM {RETURNS SYM=SYMBOL,INDIRECT,LINE END};
var
    K : integer;
begin
    SYM:=UNDEFINED;
    while(CC<LL)and(LINE[CC+1] = ' ')do CC:=CC+1;
    if(CC = LL)or(LINE[CC+1] = '$')then SYM:=LNEND
    else
        if LINE[CC+1] in ['A'..'Z','0'..'9','.',',','*','+',
            '-','!'] then
            begin
                SYM:=SYMBOL; K:=0;
                repeat
                    if K<5 then
                        begin
                            K:=K+1; ACK:=LINE [CC+1]
                        end;
                    CC:=CC+1;
                    if ACK in ['*','+', '-'] then
                        SYM:=INDIRECT;
                until(CC=LL)or(LINE[CC+1]=' ');
                while K<5 do
                    begin
                        K:=K+1; ACK:=' '
                    end
                end
            end
        else
            begin
                C:='ERR1'; ERROR
            end
        end;
    end;
{END OF GETSYM}
ocedure GETCODE {USED IN PASS2};
var
    K : integer;
begin
    for K:=1 to 4 do
        begin
            READ(PAS1,X); CLK:=X
        end
    end;
ocedure LOOKUP {LABEL TABLE IS SEARCHED};
var
    K : integer;
begin
    INTABLE:=false; K:=0;
    while(K<=I)and(not INTABLE)do
        begin

```



```

        K:= K+ 1;
        if(AC1]=LTBL[K,1])and(AC2]=LTBL[K,2])then
            INTABLE:=true
        end;
        if INTABLE then JMPADR:=VALUE [K]
    end;
Procedure INSERT {INSERT A NEW LABEL};
begin
    I:=I+1;
    if I<=20 then
        begin
            LTBL[I,1]:=AC1];LTBL[I,2]:=AC2];
            VALUE [I]:=LC
        end
    else
        begin
            C:='ERR2'; ERROR
        end
    end;
Procedure PLABEL {PROCESS THE LABEL};
begin
    if(AC1] in ['A'..'Z'])and(AC2] in ['0'..'9'])then
        begin
            LOOKUP;
            if not INTABLE then INSERT
        else
            begin
                C:='ERR4'; ERROR
            end
        end
    else
        begin
            C:='ERR3'; ERROR
        end
    end;
Procedure GETNUM {B IS VALUE OF THE NUMBER};
var
    K : integer;
begin
    B:=0;
    for K:=1 to 5 do
        begin
            if ACK] in['A'..'Z','*','+','_',' ':'']then SYNT
            if ACK] <> ' ' then
                B:=B*10 +HEXTOINT(ACK])
        end
    end;
Procedure PINDIRECT {INDIRECT ADDRESS};
begin
    if(A = '*' )then CC3]:='8'
    else

```

```

        if(A = '*+' )then CC3J:='A'
        else
            if(A = '*-' )then CC3J:='9'
            else
                begin
                    C:='ERR8'; ERROR
                end
            end;
    end;
procedure PDMA {DIRECT MEMORY ADDRESS};
    begin
        GETNUM;
        if(B<128)then
            begin
                CC3J:=INTTOHEX(B div 16);
                CC4J:=INTTOHEX(B mod 16)
            end
        else
            begin
                C:='ERR9';ERROR
            end
        end;
    end;
procedure PSHIFT;
    begin
        GETNUM;
        if B<16 then CC2J:=INTTOHEX(B)
        else
            begin
                C:='ERRA'; ERROR
            end
        end;
    end;
procedure PARP;
    begin
        GETNUM;
        if B>2 then
            begin
                C:='ERRB'; ERROR
            end
        else CC4J:=INTTOHEX(B)
    end;
procedure PPA;
    begin
        if A='PA0 ' then B:=0
        else
            if A='PA1 ' then B:=1
            else
                if A='PA2 ' then B:=2
                else
                    if A='PA3 ' then B:=3
                    else
                        if A='PA4 ' then B:=4
                        else
                            if A='PA5 ' then B:=5

```

```

else
  if A='PA6 ' then B:=6
  else
    if A='PA7 ' then B:=7
    else
      begin
        C:='ERRD';ERROR
      end
    end;
end;

Procedure PAR;
begin
  if(A= 'ARO ')then B:= 0
  else
    if(A = 'AR1 ')then B:=1
    else
      begin
        C:= 'ERRE'; ERROR
      end
    end;
end;

Procedure CODETABLE;
begin
  TABLE[1].CMD:='ABS ' ;
  TABLE[2].CMD:='ADD ' ;
  TABLE[3].CMD:='ADDD ' ;
  TABLE[4].CMD:='ADDS ' ;
  TABLE[5].CMD:='AND ' ;
  TABLE[6].CMD:='APAC ' ;
  TABLE[7].CMD:='B ' ;
  TABLE[8].CMD:='BANZ ' ;
  TABLE[9].CMD:='BGEZ ' ;
  TABLE[10].CMD:='BGZ ' ;
  TABLE[11].CMD:='BIOZ ' ;
  TABLE[12].CMD:='BLEZ ' ;
  TABLE[13].CMD:='BLZ ' ;
  TABLE[14].CMD:='BNZ ' ;
  TABLE[15].CMD:='BV ' ;
  TABLE[16].CMD:='BZ ' ;
  TABLE[17].CMD:='CALA ' ;
  TABLE[18].CMD:='CALL ' ;
  TABLE[19].CMD:='DINT ' ;
  TABLE[20].CMD:='DMOV ' ;
  TABLE[21].CMD:='EINT ' ;
  TABLE[22].CMD:='IN ' ;
  TABLE[23].CMD:='LAC ' ;
  TABLE[24].CMD:='LACK ' ;
  TABLE[25].CMD:='LAR ' ;
  TABLE[26].CMD:='LARK ' ;
  TABLE[27].CMD:='LARP ' ;
  TABLE[28].CMD:='LDP ' ;
  TABLE[29].CMD:='LDPK ' ;
  TABLE[30].CMD:='LST ' ;
  TABLE[31].CMD:='LT ' ;

```

```

TABLE[32].CMD:='LTA  ' ;
TABLE[33].CMD:='LTD  ' ;
TABLE[34].CMD:='MAR  ' ;
TABLE[35].CMD:='MPY  ' ;
TABLE[36].CMD:='MPYK ' ;
TABLE[37].CMD:='NOP  ' ;
TABLE[38].CMD:='OR   ' ;
TABLE[39].CMD:='ORG  ' ;
TABLE[40].CMD:='OUT  ' ;
TABLE[41].CMD:='PAC  ' ;
TABLE[42].CMD:='POP  ' ;
TABLE[43].CMD:='PUSH ' ;
TABLE[44].CMD:='RET  ' ;
TABLE[45].CMD:='ROVM ' ;
TABLE[46].CMD:='SACH ' ;
TABLE[47].CMD:='SACL ' ;
TABLE[48].CMD:='SAR  ' ;
TABLE[49].CMD:='SOVM ' ;
TABLE[50].CMD:='SPAC ' ;
TABLE[51].CMD:='SST  ' ;
TABLE[52].CMD:='SUB  ' ;
TABLE[53].CMD:='SUBC ' ;
TABLE[54].CMD:='SUBH ' ;
TABLE[55].CMD:='SUBS ' ;
TABLE[56].CMD:='TBLR ' ;
TABLE[57].CMD:='TBLW ' ;
TABLE[58].CMD:='XOR  ' ;
TABLE[59].CMD:='ZAC  ' ;
TABLE[60].CMD:='ZALH ' ;
TABLE[61].CMD:='ZALS ' ;
TABLE[62].CMD:='ZEQ  ' ;

end;
procedure WRITECODE {CODE IN PAS1 FILE};
begin
  WRITE(PAS1, C, ' ',
        INTTOHEX(LC div 256),
        INTTOHEX((LC div 16) mod 16),
        INTTOHEX(LC mod 16));
end;
procedure WRITETEXT {COPY SOURCE PROGRAM};
var
  K : integer;
begin
  WRITE(PAS1, M:6, ' ');
  for K:=1 to LL do
    WRITE(PAS1, LINE[K]);
  WRITELN(PAS1); M:=M+1
end;
procedure SDMA {SHIFT AND DATA MEMORY ADDRESS};
begin
  GETSYM; INDRCT:=true;
  if (SYM = LNEND) then SYNTAX;

```

```

    if(SYM = INDIRECT)then FINDIRECT
    else
        begin
            INDRCT:=false;PDMA
        end;
    GETSYM;
    if(SYM = LNEND)then
        begin
            CC2]:='0';
            if INDRCT then CC4]:='8'
        end
    else PSHIFT;
    GETSYM;
    if(SYM = LNEND)and INDRCT then CC4]:='8';
    if(SYM <> LNEND)and not INDRCT then SYNTAX;
    if(SYM <> LNEND)and INDRCT then
        begin
            PARP; GETSYM;
            if(SYM <> LNEND)then SYNTAX
        end;
    WRITECODE; WRITETEXT; LC:=LC+1
end;
Procedure DMA(DATA MEMORY ADDRESS);
begin
    GETSYM; INDRCT:=true;
    if(SYM = LNEND)then SYNTAX;
    if(SYM = INDIRECT)then FINDIRECT
    else
        begin
            INDRCT:=false; PDMA
        end;
    GETSYM;
    if(SYM = LNEND)and INDRCT then CC4]:='8';
    if(SYM <> LNEND)and not INDRCT then SYNTAX;
    if(SYM <> LNEND)and INDRCT then
        begin
            PARP; GETSYM;
            if(SYM <> LNEND)then SYNTAX
        end;
    WRITECODE; WRITETEXT; LC:=LC+1
end;
Procedure BRANCH(BRANCH ADDRESS);
begin
    WRITECODE; WRITETEXT;GETSYM;
    if(SYM = LNEND)then SYNTAX;
    if(AC1] in ['A'..'Z'])and(AC2] in ['0'..'9'])then
        begin
            LOOKUP;
            if INTABLE then
                begin
                    CC1]:='0';
                    CC2]:=INTTOHEX(JMPADR div 256);

```

```

        CC3]:=INTTOHEX((JMPADR div 16)mod 16);
        CC4]:=INTTOHEX(JMPADR mod 16)
    end
else
    begin
        CC1]:='*';   CC2]:=AC1];
        CC3]:=AC2];
        CC4]:=' '
    end
end else begin C:='ERR7'; ERROR end;
GETSYM;
if (SYM <> LNEND) then SYNTAX;
WRITECODE; WRITELN(PAS1); LC:=LC+2
end;
procedure PABS;
begin
    GETSYM;
    if SYM = LNEND then C:= '7F88'
    else SYNTAX;
    WRITECODE; WRITETEXT; LC:=LC+1
end;
procedure PADD;
begin
    CC1]:='0'; SDMA
end;
procedure PADDH;
begin
    CC1]:='6';   CC2]:='0';
    DMA
end;
procedure PADDS;
begin
    CC1]:='6';   CC2]:='1';
    DMA
end;
procedure PAND;
begin
    CC1]:='7';   CC2]:='9';
    DMA
end;
procedure PAPAC;
begin
    GETSYM;
    if SYM = LNEND then C:= '7F8F'
    else SYNTAX;
    WRITECODE; WRITETEXT; LC:=LC+1
end;
procedure PB;
begin
    C:='F900'; BRANCH
end;
procedure PBANZ;

```

```

    begin
        C:='F400'; BRANCH
    end;
procedure PBGEZ;
    begin
        C:='FD00'; BRANCH
    end;
procedure PBGZ;
    begin
        C:='FC00'; BRANCH
    end;
procedure PBIOZ;
    begin
        C:='F600'; BRANCH
    end;
procedure PBLEZ;
    begin
        C:='FB00'; BRANCH
    end;
procedure PBLZ;
    begin
        C:='FA00'; BRANCH
    end;
procedure PBNZ;
    begin
        C:='FE00'; BRANCH
    end;
procedure PBV;
    begin
        C:='F500'; BRANCH
    end;
procedure PBZ;
    begin
        C:='FF00'; BRANCH
    end;
procedure PCALA;
    begin
        GETSYM;
        if SYM = LNEND then C:= '7F8C'
        else SYNTAX;
        WRITECODE;WRITETEXT; LC:=LC+1
    end;
procedure PCALL;
    begin
        C:='F800'; BRANCH
    end;
procedure PDINT;
    begin
        GETSYM;
        if SYM = LNEND then C:= '7F81'
        else SYNTAX;
        WRITECODE; WRITETEXT; LC:=LC+1
    end;

```

```

    end;
Procedure PDMOV;
begin
    CC1J:='6';    CC2J:='9';
    DMA
end;
Procedure PEINT;
begin
    GETSYM;
    if SYM = LNEND then C:= '7F82'
    else SYNTAX;
    WRITECODE; WRITETEXT; LC:=LC+1
end;
Procedure PIN;
begin
    INDRCT:=true;    CC1J:='4';
    GETSYM;
    if(SYM = LNEND)then SYNTAX;
    if(SYM = INDIRECT)then PINDIRECT
    else
        begin
            INDRCT:=false; PDMA
        end;
    GETSYM;
    if(SYM = LNEND)then SYNTAX
    else
        begin
            PPA;    CC2J:=INTTOHEX(B)
        end;
    GETSYM;
    if(SYM = LNEND)and INDRCT then CC4J:='8';
    if(SYM <> LNEND)and not INDRCT then SYNTAX;
    if(SYM <> LNEND)and INDRCT then
        begin
            PARF;    GETSYM;
            if(SYM <> LNEND)then SYNTAX
        end;
    WRITECODE;WRITETEXT; LC :=LC+1
end;
Procedure PLAC;
begin
    CC1J:='2'; SDMA
end;
Procedure PLACK;
begin
    GETSYM;
    if(SYM <> LNEND)then GETNUM
    else SYNTAX;
    if B<256 then
        begin
            CC1J:='7';
            CC2J:='E';

```



```

        CC3]:=INTTOHEX(B div 16);
        CC4]:=INTTOHEX(B mod 16)
    end
    else SYNTAX;
    GETSYM;
    if(SYM <> LNEND)then SYNTAX;
    WRITECODE; WRITETEXT; LC:=LC+1
end;
Procedure PLAR;
begin
    GETSYM;
    if(SYM <> LNEND)or(SYM <> INDIRECT)then PAR
    else SYNTAX;
    CC1]:='3';CC2]:= INTTOHEX(B+8); DMA
end;
Procedure FLARK;
begin
    GETSYM;
    if(SYM =SYMBOL)then PAR
    else SYNTAX;
    CC1]:='7';
    CC2]:=INTTOHEX(B);
    GETSYM;
    if(SYM <> LNEND)then GETNUM
    else SYNTAX;
    if B<256 then
        begin
            CC3]:=INTTOHEX(B div 16);
            CC4]:=INTTOHEX(B mod 16)
        end
    else SYNTAX;
    GETSYM;
    if(SYM <> LNEND)then SYNTAX;
    WRITECODE; WRITETEXT; LC:=LC+1
end;
Procedure PLARP;
begin
    GETSYM;
    if(SYM <> LNEND)then GETNUM
    else SYNTAX;
    if B<2 then
        begin
            CC1]:='6';
            CC2]:='8';
            CC3]:='8';
            CC4]:=INTTOHEX(B)
        end
    else SYNTAX;
    GETSYM;
    if(SYM <> LNEND)then SYNTAX;
    WRITECODE; WRITETEXT; LC:=LC+1
end;

```

```

procedure PLDP;
begin
    CC1]:='6';    CC2]:='F';
    DMA
end;
procedure PLDPK;
begin
    GETSYM;
    if(SYM <> LNEND)then GETNUM
    else SYNTAX;
    if B<2 then
        begin
            CC1]:='6';
            CC2]:='E';
            CC3]:='0';
            CC4]:=INTTOHEX(B)
        end
    else SYNTAX;
    GETSYM;
    if(SYM <> LNEND)then SYNTAX;
    WRITECODE; WRITETEXT; LC:=LC+1
end;
procedure PLST;
begin
    CC1]:='7';    CC2]:='B'; DMA
end;
procedure PLT;
begin
    CC1]:='6';    CC2]:='A';
    DMA
end;
procedure PLTA;
begin
    CC1]:='6';    CC2]:='C';
    DMA
end;
procedure PLTD;
begin
    CC1]:='6';    CC2]:='B';
    DMA
end;
procedure PMAR;
begin
    CC1]:='6';    CC2]:='8';
    GETSYM;
    if(SYM = INDIRECT)then PINDIRECT
    else SYNTAX;
    GETSYM;
    if(SYM = LNEND)then CC4]:='8'
    else
        begin
            FARP;    GETSYM;

```

```

        if(SYM <> LNEND)then SYNTAX
      end;
      WRITECODE; WRITETEXT; LC:=LC+1
    end;
  Procedure PMPY;
  begin
    CC1]:='6'; CC2]:='D';
    DMA
  end;
  Procedure PMPYK;
  var
    K :integer; POSITIVE :boolean;
  begin
    GETSYM; POSITIVE:=true;
    if AC1] = '-' then
      begin
        POSITIVE:=false; GETSYM
      end;
    if(SYM = SYMBOL)then GETNUM
    else SYNTAX;
    if(POSITIVE and(B>4095))or
    (not POSITIVE and(B>4096))then
      begin
        C:='ERRC'; ERROR
      end;
    if not POSITIVE then B:=(8192 - B);
    K:=B div 4096;
    CC1]:=INTTOHEX(8+K);
    CC2]:=INTTOHEX((B div 256)mod 16);
    CC3]:=INTTOHEX((B div 16)mod 16);
    CC4]:=INTTOHEX(B mod 16);
    GETSYM;
    if(SYM <> LNEND)then SYNTAX;
    WRITECODE; WRITETEXT; LC:=LC+1
  end;
  Procedure FNOP;
  begin
    GETSYM;
    if SYM = LNEND then C:= '7F80'
    else SYNTAX;
    WRITECODE; WRITETEXT; LC:=LC+1
  end;
  Procedure POR;
  begin
    CC1]:='7'; CC2]:='A';
    DMA
  end;
  Procedure PORG;
  begin
    GETSYM;
    if(SYM = LNEND)then SYNTAX;
    GETNUM;

```

```

        if(B >4095)then
            begin
                C:='ERRC';    ERROR
            end;
        LC:=B;
        CC1]:='*';
        CC2]:=INTTOHEX(B div 256);
        CC3]:=INTTOHEX((B div 16)mod 16);
        CC4]:=INTTOHEX(B mod 16);
        WRITECODE; WRITETEXT
    end;
Procedure POUT;
begin
    INDRCT:=true;    CC1]:='4';
    GETSYM;
    if(SYM = LNEND)then SYNTAX;
    if(SYM = INDIRECT)then PINDIRECT
    else
        begin
            INDRCT:=false;
            PDMA
        end;
    GETSYM;
    if(SYM = LNEND)then SYNTAX
    else
        begin
            PPA;
            CC2]:=INTTOHEX(B+8)
        end;
    GETSYM;
    if(SYM = LNEND)and INDRCT then CC4]:='8';
    if(SYM <> LNEND)and not INDRCT then SYNTAX;
    if(SYM <> LNEND)and INDRCT then
        begin
            PARP;    GETSYM;
            if(SYM <> LNEND)then SYNTAX
        end;
    WRITECODE; WRITETEXT; LC:=LC+1
end;
Procedure PPAC;
begin
    GETSYM;
    if SYM = LNEND then C:= '7F8E'
    else SYNTAX;
    WRITECODE; WRITETEXT; LC:=LC+1
end;
Procedure PPOP;
begin
    GETSYM;
    if SYM = LNEND then C:= '7F9D'
    else SYNTAX;
    WRITECODE; WRITETEXT; LC:=LC+1
end;

```

```

    end;
Procedure PPUSH;
begin
    GETSYM;
    if SYM = LNEND then C:= '7F9C'
    else SYNTAX;
    WRITECODE; WRITETEXT; LC:=LC+1
end;
Procedure PRET;
begin
    GETSYM;
    if SYM = LNEND then C:= '7F8D'
    else SYNTAX;
    WRITECODE; WRITETEXT; LC:=LC+1
end;
Procedure PROVIM;
begin
    GETSYM;
    if SYM = LNEND then C:= '7F8A'
    else SYNTAX;
    WRITECODE; WRITETEXT; LC:=LC+1
end;
Procedure PSACH;
begin
    INDRCT:=true;   CL1:='5';
    GETSYM;
    if(SYM = LNEND)then SYNTAX;
    if(SYM = INDIRECT)then PINDIRECT
    else
        begin
            INDRCT:=false;
            PDMA
        end;
    GETSYM;
    if(SYM = LNEND)then
        begin
            CL2:='8';
            if INDRCT then CL4:='8'
        end
    else
        begin
            PSHIFT;
            if(B=0)or(B = 1)or(B = 4)then
                begin
                    B:=B+8;   CL2:=INTTOHEX(B);
                end
            else
                begin
                    C:='ERRA'; ERROR
                end;
            GETSYM;
            if(SYM=LNEND)and INDRCT then CL4:='8';

```

```

        if(SYM<>LNEND)and not INDRCT then SYNTAX;
        if(SYM <> LNEND)and INDRCT then
            begin
                PARP;  GETSYM;
                if(SYM <> LNEND)then SYNTAX
            end
        end;
        WRITECODE; WRITETEXT; LC:=LC+1
    end;
Procedure PSACL;
begin
    INDRCT:=true; CC1]:='5';
    GETSYM;
    if(SYM = LNEND)then SYNTAX;
    if(SYM = INDIRECT)then PINDIRECT
    else
        begin
            INDRCT:=false;
            PDMA
        end;
    GETSYM;
    if(SYM = LNEND)then
        begin
            CC2]:='0';
            if INDRCT then CC4]:='8'
        end
    else
        begin
            PSHIFT;
            if(B<>0)then
                begin
                    C:='ERRA'; ERROR
                end;
            GETSYM;
            if(SYM=LNEND)and INDRCT then CC4]:='8';
            if(SYM<>LNEND)and not INDRCT then SYNTAX;
            if(SYM <> LNEND)and INDRCT then
                begin
                    PARP;  GETSYM;
                    if(SYM <> LNEND)then SYNTAX
                end
            end;
            WRITECODE; WRITETEXT; LC:=LC+1
        end;
Procedure PSAR;
begin
    GETSYM;
    if(SYM <> LNEND)or(SYM <> INDIRECT)then PAR
    else SYNTAX;
    CC1]:='3';
    CC2]:=INTTOHEX(B);
    GETSYM;  INDRCT:=true;

```

```

    if(SYM = LNEND)then SYNTAX;
    if(SYM = INDIRECT)then PINDIRECT
    else
        begin
            INDRCT:=false;
            PDMA
        end;
    GETSYM;
    if(SYM = LNEND)and INDRCT then CC4]:='8';
    if(SYM <> LNEND)and not INDRCT then SYNTAX;
    if(SYM <> LNEND)and INDRCT then
        begin
            PARP; GETSYM;
            if(SYM <> LNEND)then SYNTAX
        end;
    WRITECODE; WRITETEXT; LC:=LC+1
end;
Procedure PSOVM;
begin
    GETSYM;
    if SYM = LNEND then C:= '7F8B'
    else SYNTAX;
    WRITECODE; WRITETEXT; LC:=LC+1
end;
Procedure PSPAC;
begin
    GETSYM;
    if SYM = LNEND then C:= '7F90'
    else SYNTAX;
    WRITECODE; WRITETEXT; LC:=LC+1
end;
Procedure PSST;
begin
    CC1]:='7'; CC2]:='C';
    DMA
end;
Procedure PSUB;
begin
    CC1]:='1'; SDMA
end;
Procedure PSUBC;
begin
    CC1]:='6'; CC2]:='4';
    DMA
end;
Procedure PSUBH;
begin
    CC1]:='6'; CC2]:='2';
    DMA
end;
Procedure PSUBS;
begin

```

```

        CC1]:= '6';    CC2]:= '3';
        DMA
    end;
Procedure PTBLR;
begin
    CC1]:= '6';    CC2]:= '7';
    DMA
end;
Procedure PTBLW;
begin
    CC1]:= '7';    CC2]:= 'D';
    DMA
end;
Procedure PXOR;
begin
    CC1]:= '7';    CC2]:= '8';
    DMA
end;
Procedure PZAC;
begin
    GETSYM;
    if SYM = LNEND then C:= '7F89'
    else SYNTAX;
    WRITECODE; WRITETEXT; LC:=LC+1
end;
Procedure PZALH;
begin
    CC1]:= '6';    CC2]:= '5';
    DMA
end;
Procedure PZALS;
begin
    CC1]:= '6';    CC2]:= '6';
    DMA
end;
Procedure PZEQ;
var
    POSITIVE , ONE : boolean;
begin
    GETSYM; ONE:=true;
    if SYM = LNEND then SYNTAX;
    while SYM <> LNEND do
        begin
            POSITIVE:= true;
            if AC1] = '-' then
                begin
                    POSITIVE:=false;    GETSYM
                end;
            if SYM = SYMBOL then GETNUM
            else SYNTAX;
            if not POSITIVE then B:=(65536 - B);
            CC1]:=INTTOHEX(B div 4096);

```



```

        CC2]:=INTTOHEX((B div 256)mod 16);
        CC3]:=INTTOHEX((B div 16)mod 16);
        CC4]:=INTTOHEX(B mod 16);
        WRITECODE;
        if ONE then WRITETEXT
        else WRITELN(PAS1);
        ONE:=false; GETSYM; LC:=LC+1
    end
end;
Procedure GETCOMMAND (SELECTS PROPER PROCEDURE);
var
    DONE :boolean;
    FIRST, LAST, MIDDLE : integer;
begin
    DONE:=false; COMMAND:=A;
    FIRST:=1; LAST:=62;
    while((LAST >= FIRST)and(not DONE))do
        begin
            MIDDLE:=(FIRST+LAST)div 2;
            if COMMAND > TABLE[MIDDLE].CMD then
                FIRST:=MIDDLE + 1
            else
                if COMMAND < TABLE[MIDDLE].CMD then
                    LAST:=MIDDLE - 1
                else
                    if COMMAND=TABLE[MIDDLE].CMD then
                        DONE:=true
                    end;
                end;
            if DONE then
                case MIDDLE of
                    1 : PABS;
                    2 : PADD;
                    3 : PADDH;
                    4 : PADDL;
                    5 : PAND;
                    6 : PAPAC;
                    7 : PB;
                    8 : PBANZ;
                    9 : PBGEZ;
                    10 : PBGZ;
                    11 : PBIOZ;
                    12 : PBLEZ;
                    13 : PBLZ;
                    14 : PENZ;
                    15 : PBV;
                    16 : PBZ;
                    17 : PCALA;
                    18 : PCALL;
                    19 : PDINT;
                    20 : PDMOV;
                    21 : PEINT;
                    22 : FIN;

```

```

23 : PLAC;
24 : PLACK;
25 : PLAR;
26 : PLARK;
27 : PLARP;
28 : PLDP;
29 : PLDPK;
30 : PLST;
31 : PLT;
32 : PLTA;
33 : PLTD;
34 : PMAR;
35 : PMPY;
36 : PMPYK;
37 : PNOP;
38 : POR;
39 : FORG;
40 : FOUT;
41 : FPAC;
42 : FPOP;
43 : FPUSH;
44 : FRET;
45 : PROVM;
46 : PSACH;
47 : PSACL;
48 : PSAR;
49 : PSOVM;
50 : PSPAC;
51 : PSST;
52 : PSUB;
53 : PSUBC;
54 : PSUBH;
55 : PSUBS;
56 : PTBLR;
57 : PTBLW;
58 : PXOR;
59 : PZAC;
60 : PZALH;
61 : PZALS;
62 : PZEQ

end
else
    besin
        C:='ERR6'; ERROR
    end
end;
{MAIN PROGRAM BEGINS}
{INPUT-OUTPUT FILE DECLARATION}
besin
    WRITE(TTY,'GIVE INPUT FILE NAME : '); BREAK(TTY);
    I:=0;
    while not EOLN(TTY)do GET(TTY);

```

```

GET(TTY);
while not EOLN(TTY) do
begin
    I:=I+1;
    if I <= 6 then NINP[I]:=TTY;
    GET(TTY)
end;
while I<6 do
begin
    I:=I+1; NINP[I]:=' '
end;
NINP[7]:='I';
NINP[8]:='N';
NINP[9]:='P';
RESET(FINP,NINP);
WRITE(TTY,'GIVE OUTPUT FILE NAME : '); BREAK(TTY);
I:=0;
while not EOLN(TTY)do GET(TTY);
GET(TTY);
while not EOLN(TTY)do
begin
    I:=I+1;
    if I<=6 then NOUP[I]:=TTY;
    GET(TTY)
end;
while I<6 do
begin
    I:=I+1; NOUP[I]:=' '
end;
NOUP[7]:='D';
NOUP[8]:='B';
NOUP[9]:='J';
REWRITE(FOUP,NOUP);
{PASS ONE BEGINS}
I:=0; M:=0;{LABEL AND LINE # INITIALIZED}
OK:=true;REWRITE(PAS1);{OK FLAG & PAS1 FILE INITIALIZED}
M:= M+1;
LC:=0;
CODETABLE;
repeat
    GETLINE; GETSYM;
    if(SYM = SYMBOL)and(LINE[3] = ':')then
        begin
            PLABEL;
            GETSYM
        end;
    if SYM <> LNEND then GETCOMMAND
    else SYNTAX;
    100 ;;
    READLN(FINP)
until EOF(FINP);
{PASS TWO BEGINS}

```

```

if OK then
begin
  RESET(PAS1);
  while not EOF(PAS1)do
  begin
    GETCODE;
    if(CC1] = '*' )and(CC4] = ' ' )then
    begin
      AC1]:=CC2];
      AC2]:=CC3];
      LOOKUP;
      if INTABLE then
      begin
        CC1]:= '0';
        CC2]:=INTTOHEX(JMPADR div 256);
        CC3]:=INTTOHEX((JMPADR div 16)mod 16);
        CC4]:=INTTOHEX(JMPADR mod 16);
      end
      else
        C:= 'ERRF'
      end;
    WRITE(FOUP,C);
    while not EOLN(PAS1)do
    begin
      READ(PAS1,X);  WRITE(FOUP,X)
    end;
    READLN(PAS1);  WRITELN(FOUP)
  end
end
end
end .

```

APPENDIX DSIMULATOR PROGRAM LISTING

```

PROGRAM SIMULATOR(TTY,FOBJ,INPUT,OUTPUT);
const
    KING=4294967296;(2**32)
    QUEEN=2147483648;(2**31)
    ROOK=65536;(2**16)
    KNIGHT=32768;(2**15)
    PAWN=4096;(2**15)
    NBRK=8 { SIZE OF THE BREAK-POINT TABLE };
type
    WRD=0..65535;
    LOCATION=0..4095;
    CMDARRAY=packed array [1..8] of char;
    Y=record
        CMD :CMDARRAY
    end;
var
    PMEMORY : array [0..4095] of WRD;{program memory}
    DMEMORY : array [0..143] of WRD;{data memory}
    INPORT : array [0..7] of WRD;
    OUTPORT : array [0..7] of WRD;
    IR,AR0,AR1,TR :WRD;
    IRH,IRL : 0..255;
    PC,LOW,HIGH : 0..4095;
    STACK : array [0..3] of 0..4095;
    BRKPT : packed array [0..4095] of boolean {BRKPT FLAGS};
    BRKTABLE : array [0..NBRK] of record {table of break-points
        FLAG : boolean;
        PLACE : 0..4095
    end;
    ERRFLAG : boolean { unimplemented instruction flag };
    TRFLG,FLGTR : boolean {TRACE FLAG};
    OVM : 0..1;
    CK : 0..5000000{CLOCK CYCLES};
    BIO,OVFL,INTM,ARP,DP :0..1;
    ACC,PR,SCRATCH,TEMP : 0..4294967295;
    SYM :(IDENTIFIER,NUMBER,SLASH,LNEND,UNDEFINED);
    A,COMMAND:packed array [1..8] of char{lexeme/command};
    TABLE : array [1..20] of Y ;
    B : integer {numeric lexeme };
    LINE : array [1..81] of char {command line };
    CC : integer {character count };
    LL : integer {line length };
    RATE : integer{USER DEFINED INTERRUPT INTERVAL};
    INTRFLG : boolean{INTERRUPT ACTIVE FLAG};
    FOBJ : TEXT;
    NOBJ : packed array [1..9] of char;
function INTTOHEX(N:integer): char;

```

```

PROGRAM SIMULATOR(TTY,FOBJ,INPUT,OUTPUT);
const
  KING=4294967296;(2**32)
  QUEEN=2147483648;(2**31)
  ROOK=65536;(2**16)
  KNIGHT=32768;(2**15)
  PAWN=4096;(2**15)
  NBRK=8 { SIZE OF THE BREAK-POINT TABLE };
type
  WRD=0..65535;
  LOCATION=0..4095;
  CMDARRAY=packed array [1..8] of char;
  Y=record
    CMD :CMDARRAY
  end;
var
  PMEMORY : array [0..4095] of WRD;(program memory)
  DMEMORY : array [0..143] of WRD;(data memory)
  INPORT : array [0..7] of WRD;
  OUTPORT : array [0..7] of WRD;
  IR,ARO,ARI,TR :WRD;
  IRH,IRL : 0..255;
  PC,LOW,HIGH : 0..4095;
  STACK : array [0..3] of 0..4095;
  BRKPT : packed array [0..4095] of boolean (BRKPT FLAGS);
  BRKTABLE : array [0..NBRK] of record {table of break-points
    FLAG : boolean;
    PLACE : 0..4095
  end;
  ERRFLAG : boolean { unimplemented instruction flag };
  TRFLG,FLGTR : boolean {TRACE FLAG};
  OVM : 0..1;
  CK : 0..50000000(CLOCK CYCLES);
  BIO,OVFL,INTM,ARP,DP :0..1;
  ACC,PR,SCRATCH,TEMP : 0..4294967295;
  SYM : (IDENTIFIER,NUMBER,SLASH,LNEND,UNDEFINED);
  A,COMMAND:packed array [1..8] of char(lexeme/command);
  TABLE : array [1..20] of Y ;
  B : integer {numeric lexeme };
  LINE : array [1..81] of char {command line };
  CC : integer {character count };
  LL : integer {line length };
  RATE : integer{USER DEFINED INTERRUPT INTERVAL};
  INTRFLG : boolean{INTERRUPT ACTIVE FLAG};
  FOBJ : TEXT;
  NOBJ : packed array [1..9] of char;
function INTTOHEX(N:integer): char;
var
  T : char;
begin
  case N of
    0 : T:='0';

```

```

        1 : T:='1';
        2 : T:='2';
        3 : T:='3';
        4 : T:='4';
        5 : T:='5';
        6 : T:='6';
        7 : T:='7';
        8 : T:='8';
        9 : T:='9';
       10 : T:='A';
       11 : T:='B';
       12 : T:='C';
       13 : T:='D';
       14 : T:='E';
       15 : T:='F';
    end;
    INTTOHEX:=T
  end { internal-to-hex };
function HEXTOINT(H:char): integer;
var
    T : integer;
begin
    case H of
        '0' : T:=0;
        '1' : T:=1;
        '2' : T:=2;
        '3' : T:=3;
        '4' : T:=4;
        '5' : T:=5;
        '6' : T:=6;
        '7' : T:=7;
        '8' : T:=8;
        '9' : T:=9;
        'A' : T:=10;
        'B' : T:=11;
        'C' : T:=12;
        'D' : T:=13;
        'E' : T:=14;
        'F' : T:=15;
    end;
    HEXTOINT:=T
  end { hex-to-internal };
procedure INIT(RESETS ALL EXCEPT PROG MEM & OVFL);
var
    K :integer;
begin
    for K:=0 to 143 do DMEMORY[K]:=0;
    for K:=0 to 7 do
        begin
            INPORT[K]:=0; OUTPORT[K]:=0;
        end;
    for K:=0 to 3 do STACK[K] :=0;

```

```

    ACC:=0; PR:=0; PC:=0; ARO:=0; AR1:=0;
    TR:=0; ARP:=0; DP:=0; OVM:=0; BIO:=1;
    INTM :=0; CK:=0
end;
Procedure LOAD; (PROGRAM, CONSTANTS & DATA IN PMEMORY)
var
    I : integer;
    PLOC : 0..4095;
    X : char;
begin
    WRITE(TTY, 'GIVE OBJ FILENAME : ');
    BREAK(TTY); I:=0;
    while not EOLN(TTY) do GET(TTY);
    GET(TTY);
    while not EOLN(TTY) do
        begin
            I:=I+1;
            if I<=6 then NOBJC11:=TTY^;
            GET(TTY)
        end;
    while I<6 do
        begin
            I:=I+1; NOBJC11:=' '
        end;
    NOBJC7:= '0'; NOBJC8:= 'B'; NOBJC9:= 'J';
    RESET(FOBJ, NOBJ); B:=0;
    for I:=1 to 4 do
        begin
            READ(FOBJ, X); AC11:=X
        end;
    for I:=2 to 4 do B:=B*16 +HEXTOINT(AC11);
    PC:=B; PLOC:=B;
    READLN(FOBJ);
    while not EOF(FOBJ) do
        begin
            B:=0;
            for I:=1 to 4 do
                begin
                    READ(FOBJ, X); AC11:=X
                end;
            if AC11='*' then
                begin
                    for I:=2 to 4 do
                        B:=B*16+HEXTOINT(AC11);
                    PLOC:=B
                end
            else
                begin
                    for I:=1 to 4 do
                        B:=B*16 +HEXTOINT(AC11);
                    PMEMORY[PLOC]:=B;
                    PLOC:=(PLOC+1)mod 4096
                end
            end
        end
    end
end

```



```

                end;
                READLN(FOBJ)
            end
        end;
    Procedure GETSYM { lexical analyser };
    var
        K : integer;
    begin
        SYM:=UNDEFINED;
        while(CC<LL)and(LINE[CC+1]=' ')do CC:=CC+1;
        if CC=LL then SYM:=LNEND
        else
            if LINE[CC+1] in ['A'..'Z','0'..'9'] then
                begin
                    SYM:=NUMBER;    K:=0;    B:=0;
                    repeat
                        if K<8 then
                            begin
                                K:=K+1; ACK:=LINE[CC+1]
                            end;
                                CC:=CC+1;
                                if ACK in ['G'..'Z'] then
                                    SYM:=IDENTIFIER
                                until(CC=LL)or not(LINE[CC+1] in
                                    ['A'..'Z','0'..'9']);
                                while K<8 do
                                    begin    K:=K+1;    ACK:=' '
                                    end;
                                    if SYM=NUMBER then
                                        for K:=1 to 8 do
                                            if ACK<>' ' then
                                                B:=B*16+HEXTOINT(ACK)
                                        end
                                    else
                                        begin
                                            if LINE[CC+1]='/' then SYM:=SLASH;
                                            CC:=CC+1
                                        end
                                    end
                                end { setsym };
        Procedure ERROR;{COMMAND ERROR}
        begin
            WRITELN(TTY);
            WRITELN(TTY,'Command-syntax-error')
        end { error };
    Procedure GETLINE;{COPY ONE LINE}
    begin
        while not EOLN(TTY)do GET(TTY);
        GET(TTY);CC:=0;    LL:=0;
        while not EOLN(TTY)do
            begin    LL:=LL+1; LINE[LL]:=TTY^; GET(TTY)
            end
        end { getline };

```

```
Procedure DEPOSITD; (DEPOSIT IN DATA MEMORY)
```

```
var
  DLOC : 0..143;
  T : packed array [1..8] of char;
begin
  GETSYM;
  if SYM=NUMBER then
    begin
      DLOC:=B mod 144;
      repeat
        AC1:=INTTOHEX(DLOC div 16);
        AC2:=INTTOHEX(DLOC mod 16);
        WRITE(TTY,AC1,AC2,'/');
        BREAK(TTY);GETLINE; GETSYM;
      if SYM=NUMBER then
        begin
          while B>65535 do B:=B mod 65535;
          DMEMORY[DLOC] :=B;
          DLOC:=(DLOC+1)mod 144;
        end
      else
        if SYM<>LNEND then ERROR;
      until SYM=LNEND
    end
  else ERROR
end (DEPOSITD);
```

```
Procedure DISPLAYD; (DISPLAY DATA MEMORY)
```

```
var
  DLOC :0..143;
begin
  GETSYM;
  if SYM=NUMBER then
    begin
      DLOC:=B mod 144;
      repeat
        AC1:=INTTOHEX(DLOC div 16);
        AC2:=INTTOHEX( DLOC mod 16 );
        WRITE(TTY,AC1,AC2,'/');
        INTTOHEX(DMEMORY[DLOC] div PAWN);
        INTTOHEX((DMEMORY[DLOC] div 256)mod 16);
        INTTOHEX((DMEMORY[DLOC] div 16)mod 16);
        INTTOHEX(DMEMORY[DLOC] mod 16 );
        BREAK(TTY);GETLINE; GETSYM;
        if SYM<>SLASH then
          DLOC:=(DLOC+1)mod 144;
        until SYM=SLASH
      end
    else ERROR
  end(DISPLAY DATA MEMORY);
```

```
Procedure DEPOSITP;
```

```
var
  PLOC :0..4095;
```

```

begin
  GETSYM;
  if SYM=NUMBER then
    begin
      PLOC:=B mod 4096;
      repeat
        AC1]:=INTTOHEX(PLOC div 256);
        AC2]:=INTTOHEX((PLOC div 16)mod 16);
        AC3]:=INTTOHEX(PLOC mod 16);
        WRITE(TTY,AC1],AC2],AC3],'/');
        BREAK(TTY); GETLINE; GETSYM;
      if SYM=NUMBER then
        begin
          while B>65535 do B:=B mod 8000;
          PMEMORY[PLOC]:=B;
          PLOC:=(PLOC+1)mod PAWN;
        end
      else
        if SYM<>LNEND then ERROR;
      until SYM=LNEND
    end
  else ERROR
end{DEPOSITP};

Procedure DISPLAYP;
var
  PLOC : 0..4095;
begin
  GETSYM;
  if SYM=NUMBER then
    begin
      PLOC:=B mod PAWN;
      repeat
        AC1]:=INTTOHEX(PLOC div 256);
        AC2]:=INTTOHEX((PLOC div 16)mod 16);
        AC3]:=INTTOHEX(PLOC mod 16);
        WRITE(TTY,AC1],AC2],AC3],'/');
        INTTOHEX(PMEMORY[PLOC] div PAWN),
        INTTOHEX((PMEMORY[PLOC] div 256)mod 16),
        INTTOHEX((PMEMORY[PLOC] div 16)mod 16),
        INTTOHEX(PMEMORY[PLOC] mod 16));
        BREAK(TTY); GETLINE; GETSYM;
        if SYM<>SLASH then
          PLOC:=(PLOC+1)mod PAWN;
        until SYM=SLASH
      end
    else ERROR
  end{DISPLAY PROGRAM MEMORY}
;

Procedure DEPIPORT;
var
  PORT :0..7;
begin

```

```

GETSYM;
if SYM=NUMBER then
  begin
    PORT:=B mod 8;
    repeat
      WRITE(TTY,'I[PORT]',PORT:2,'J/');
      BREAK(TTY); GETLINE; GETSYM;
      if SYM=NUMBER then
        begin
          while B>65535 do B:=B mod 8000 ;
          INPORT[PORT]:=B;
          PORT:=(PORT+1)mod 8
        end
      else
        if SYM<>LNEND then ERROR;
      until SYM=LNEND
    end
  else ERROR
end{DEPOSIT IN PORT};
Procedure DISOPORT;
var
  PORT : 0..7;
begin
  GETSYM;
  if SYM=NUMBER then
    begin
      PORT:=B mod 8;
      repeat
        WRITE(TTY,'O[PORT]',PORT,'J/');
        INTTOHEX(OUTPORT[PORT] div PAWN),
        INTTOHEX((OUTPORT[PORT] div 256)mod 16),
        INTTOHEX((OUTPORT[PORT] div 16)mod 16),
        INTTOHEX(OUTPORT[PORT] mod 16));
        BREAK(TTY); GETLINE; GETSYM;
        if SYM<>SLASH then PORT:=(PORT+1)mod 8;
      until SYM=SLASH
    end
  else ERROR
end{DISPLAY OUT PORT};
Procedure DEPOSITR;
var
  T : packed array [1..8] of char;
begin
  GETSYM; T:=A;
  if (T='PC' ) then
    begin
      WRITE(TTY,A,'/');BREAK(TTY);GETLINE; GETSYM;
      if SYM=NUMBER then
        begin
          while B>4095 do B:=B mod PAWN;
          PC:=B;
        end
      end
    end
  end
end

```

```

else ERROR
end
else
  if(A='ARO      ')or(A='AR1      ')
  or(A='IR       ')or(A='TR       ')then
    begin
      WRITE(TTY,A,'//');
      BREAK(TTY); GETLINE; GETSYM;
      if SYM=NUMBER then
        begin
          while B>65535 do B:=B div 16;
          if T='ARO      ' then ARO:=B
          else
            if T='AR1      ' then
              AR1:=B
            else
              if T='TR       ' then
                TR:=B
              else
                if T='IR       '
                then IR:=B
            end
          end
        end
      else ERROR
    end
  else
    if(A='ACC      ')or(A='PR       ')then
      begin
        WRITE(TTY,A,'//');
        BREAK(TTY); GETLINE;GETSYM;
        if SYM=NUMBER then
          begin
            while B>=KING do
              B:=B mod KING;
            if T='ACC      ' then
              ACC:=B
            else
              if T='PR       '
              then PR:=B
            end
          end
        else ERROR
      end
    else
      if(A='ARP      ')or(A='DP       ')
      or(A='BIO      ')then
        begin
          WRITE(TTY,A,'//');
          BREAK(TTY); GETLINE;GETSYM;
          if SYM=NUMBER then
            begin
              while B>1 do
                B:=B div 2;
              if T='ARP      '

```

```

                                then ARP:=B
                                else
                                    if T='DP
                                        then DP:=B
                                    end
                                else ERROR;
                                GETSYM;
                                if SYM<>LNEND then ERROR
                                end
                                else ERROR
                                end {DEPOSIT REGISTER};
procedure DISPLAYR;{display register}
begin
    GETSYM;
    if(A='AR0      ')or(A='AR1      ')or(A='TR      ')
    or(A='IR      ')or(A='PC      ')then
        begin
            if A='PC      ' then
                begin
                    B:=PC;
                    WRITELN(TTY,A,'/',
                        INTTOHEX(B div 256)
                        ,INTTOHEX((B div 16)mod 16),
                        INTTOHEX(B mod 16));
                    BREAK(TTY)
                end
            else
                if A='AR0      ' then B:=AR0
                else
                    begin
                        if A='AR1      ' then B:=AR1
                        else
                            if A='TR      ' then B:=TR
                            else
                                if A='IR      '
                                    then B:=IR;
                                WRITELN(TTY,A,'/',
                                    INTTOHEX(B div PAWN),
                                    INTTOHEX((B div 256)mod 16),
                                    INTTOHEX((B div 16)mod 16),
                                    INTTOHEX(B mod 16));
                                BREAK(TTY)
                            end
                        end
                    end
                end
            end
        end
    else
        if(A='ACC      ')or(A='PR      ')then
            begin
                if A='ACC      ' then B:=ACC
                else
                    if A='PR      ' then B:=PR;
                    WRITELN(TTY,A,'/',
                        INTTOHEX(B div 268435456),

```

```

INTTOHEX((B div 1677216)mod 16),
INTTOHEX((B div 1048576)mod 16),
INTTOHEX((B div 65536)mod 16), ' ',
INTTOHEX((B div 4096)mod 16),
INTTOHEX((B div 256)mod 16),
INTTOHEX((B div 16)mod 16),
INTTOHEX(B mod 16));
BREAK(TTY);
end
else ERROR
end {DISPLAYR};
Procedure SETTRACE;{set trace}
begin
WRITE(TTY,'LOW : ');
BREAK(TTY); GETLINE; GETSYM;
if SYM=NUMBER then LOW:=B mod PAWN
else ERROR;
WRITE(TTY,'HIGH : ');
BREAK(TTY); GETLINE; GETSYM;
if SYM=NUMBER then HIGH:=B mod PAWN
else ERROR;
TRFLG:=true
end;
Procedure REMTRACE;{remove trace}
begin
TRFLG:=false;
FLGTR:=false
end;
Procedure PLOT;
var
HIGH,LOW,PLOC :0..4095;
begin
WRITE(TTY,'LOW : ');
BREAK(TTY); GETLINE; GETSYM;
if SYM=NUMBER then LOW:=B mod PAWN
else ERROR;
WRITE(TTY,'HIGH : ');
BREAK(TTY); GETLINE; GETSYM;
if SYM=NUMBER then HIGH:=B mod PAWN
else ERROR;
repeat
PLOC:=LOW; LOW:=LOW+2;
begin
B:=PMEMORY[PLOC];
B:=(B+ KNIGHT)mod ROOK;
B:=B div 820;
repeat
WRITE(OUTPUT,' '); B:=B-1
until B=0;
Writeln(OUTPUT,'*')
end;
until LOW>HIGH

```

```

end;
procedure PINTR{SETS INTERRUPTS AT USER DEFINED INTERVALS};
begin
  WRITE(TTY, 'GIVE INTERRUPT AT CLOCK CYCLES : ');
  BREAK(TTY); GETLINE; GETSYM;
  if SYM=NUMBER then RATE:=B
  else ERROR;
  INTRFLG:=true
end;
procedure EXECUTE { EXECUTE ONE INSTRUCTION };
var
  PREVPC : 0..4095;
  IRH, IRL : 0..255;
  PLOC : 0..4095;
  DLOC : 0..255;
  INTERRUPT : boolean{INTERRUPT FLAG};
  PREVINTR:boolean{INTERRUPT IN LAST CYCLE:FLAG};
  DELAY:boolean{INTERRUPT TO BE DELAYED BY ONE CYCLE};
procedure BADINST {bad instruction}
begin
  ERRFLAG:=true;
  WRITELN(TTY);
  WRITELN(TTY,
    'UNIMPLEMENTED INSTRUCTION AT USER PC:',
    INTTOHEX(PREVPC div 256),
    INTTOHEX((PREVPC div 16)mod 16),
    INTTOHEX(PREVPC mod 16))
  end {BADINST};
procedure DERROR{data address - warnings}
begin
  WRITELN(TTY);
  WRITELN(TTY,
    'DATA ADDRESS OUT OF RANGE AT USER PC:',
    INTTOHEX(PREVPC div 256),
    INTTOHEX((PREVPC div 16)mod 16),
    INTTOHEX(PREVPC mod 16))
  end;
procedure TRACE{TRACE DISPLAYED ON TERMINAL};
begin
  WRITELN(TTY, 'PC:',
    INTTOHEX(PC div 256),
    INTTOHEX((PC div 16)mod 16),
    INTTOHEX(PC mod 16), ' ',
    'ACC:', INTTOHEX(ACC div 268435456),
    INTTOHEX((ACC div 16777216)mod 16),
    INTTOHEX((ACC div 1048576)mod 16),
    INTTOHEX((ACC div R00K)mod 16), ' ',
    INTTOHEX((ACC div PAWN)mod 16),
    INTTOHEX((ACC div 256)mod 16),
    INTTOHEX((ACC div 16)mod 16),
    INTTOHEX(ACC mod 16), ' ',
    'ARO:', INTTOHEX(ARO div PAWN),

```



```

        INTTOHEX((AR0 div 256)mod 16),
        INTTOHEX((AR0 div 16)mod 16),
        INTTOHEX(AR0 mod 16),' ',
        'AR1:',INTTOHEX(AR1 div PAWN),
        INTTOHEX((AR1 div 256)mod 16),
        INTTOHEX((AR1 div 16)mod 16),
        INTTOHEX(AR1 mod 16),' ',
        'STATUS:',OVFL:2,OVM:2,INTM:2,
        ARP:2,DP:2,' CLK:',CK:5);
    BREAK(TTY)
end;
Procedure PINTERRUPT{CHANGES PC APPROPRIATELY};
var
    K:integer;
begin
    if(CK mod RATE=0)then
        begin INTERRUPT:=true; PREVINTR:=true
        end else
        if((CK mod RATE=1)or(CK mod RATE=2))and
        not PREVINTR then
            begin INTERRUPT:=true;PREVINTR:=true
            end
        else PREVINTR:=false;
        if(INTERRUPT and not DELAY and(INTM=0))then
            begin
                INTERRUPT:=false;INTM:=1;
                for K:=3 downto 1 do
                    STACK[K]:=STACK[K-1];
                    STACK[0]:=PC;PC:=2
                end
            end
        end;
Procedure OPRND;{fetch operand,modify AR,ARP}
    Procedure PINCREMENT;
    begin
        if ARP=0 then
            if(AR0 mod 512=511)
                then AR0:=AR0-511
            else AR0:=AR0+1
        else
            if(AR1 mod 512=511)
                then AR1:=AR1-511
            else AR1:=AR1+1
        end;
    Procedure PDECREMENT;
    begin
        if ARP=0 then
            if(AR0 mod 512=0)then AR0:=AR0+51
            else AR0:=AR0-1
        else
            if(AR1 mod 512=0)then AR1:=AR1+51
            else AR1:=AR1-1
        end;
end;

```

```

begin
  if IRL<128 then DLOC:=DP*128+IRL
  else
    begin
      if ARP=0 then DLOC:=AR0 mod 256
      else DLOC:=AR1 mod 256;
      IRL:=IRL-128;
      if IRL>=48 then BADINST;
      if (IRL mod 8>1) then BADINST;
      if (IRL div 32=1) then PINCREMENT;
      if (IRL div 16=1) then PDECREMENT;
      if (IRL mod 16<2) then ARP:=IRL mod 2
    end;
  if DLOC< 144 then
    SCRATCH:=DMEMORY [DLOC]
  else DERROR
end {FETCH OPERAND};
Procedure PSGNA ;{sign & overflow}
begin
  if ((ACC<QUEEN) and (SCRATCH<QUEEN)
    and (TEMP>=QUEEN)) then
    begin
      OVFL:=1;
      if OVM=1 then ACC:=QUEEN-1
      else ACC:=TEMP
    end
  else
    if ((ACC >=QUEEN) and (SCRATCH >=QUEEN)
      and (TEMP<QUEEN)) then
      begin
        OVFL:=1;
        if OVM=1 then ACC:=QUEEN
        else ACC:=TEMP
      end
    else ACC:=TEMP
  end;
Procedure PTRACE;{set or reset trace}
begin
  if LOW=PREVPC then FLGTR:=true;
  if HIGH=(PREVPC-1) then FLGTR:=false;
end;
Procedure PABS;
begin
  if ACC >=QUEEN then ACC:=KING-ACC;
  if (ACC=QUEEN) and (OVM=1) then
    ACC:=QUEEN-1;
  CK:=CK+1
end;
Procedure PADD;
var
  D : integer;
begin

```

```

    OPRND;
    if (SCRATCH >= KNIGHT) then
        SCRATCH := KING + SCRATCH - ROOK;
    D := IRH;
    while D > 0 do
        begin
            SCRATCH := (SCRATCH * 2) mod KING;
            D := D - 1;
        end;
    TEMP := (ACC + SCRATCH) mod KING;
    PSGNA;
    CK := CK + 1;
end;
procedure PADDH;
begin
    OPRND;
    SCRATCH := SCRATCH * ROOK;
    TEMP := (ACC + SCRATCH) mod KING;
    PSGNA;
    CK := CK + 1;
end;
procedure PADDS;
begin
    OPRND;
    TEMP := (ACC + SCRATCH) mod KING;
    PSGNA;
    CK := CK + 1;
end;
procedure PAND;
var
    K, D : integer;
begin
    OPRND;
    TEMP := ACC mod ROOK;
    ACC := 0;
    D := ROOK;
    for K := 1 to 16 do
        begin
            ACC := ACC * 2; D := D div 2;
            if ((SCRATCH div D) mod 2 = 1) and
                ((TEMP div D) mod 2 = 1) then
                ACC := ACC + 1;
        end;
        CK := CK + 1;
    end;
end;
procedure PAPAC;
begin
    SCRATCH := PR;
    TEMP := (ACC + SCRATCH) mod KING;
    PSGNA;
    CK := CK + 1;
end;

```

```

Procedure PB;
begin
  PC:=(PMEMORY[PC])mod PAWN;
  CK:=CK+2
end;
Procedure PBANZ;
begin
  if ARP=0 then
    begin
      SCRATCH:=AR0 mod 512;
      if SCRATCH<>0 then
        begin
          PLOC:=PC;
          PC:=(PMEMORY[PLOC])mod PAWN;
          AR0:=AR0-1
        end
      else
        begin
          PC:=PC+1;
          AR0:=AR0+511
        end
      end
    else
      begin
        SCRATCH:=AR1 mod 512;
        if SCRATCH<>0 then
          begin
            PLOC:=PC;
            PC:=(PMEMORY[PLOC])mod PAWN;
            AR1:=AR1-1
          end
        else
          begin
            PC:=PC+1;
            AR1:=AR1+511
          end
        end
      end;
    CK:=CK+2
  end;
Procedure PBGEZ;
begin
  if ACC< QUEEN then
    PC:=(PMEMORY[PC])mod PAWN
  else PC:=PC+1;
  CK:=CK+2
end;
Procedure PBGZ;
begin
  if (ACC>0)and(ACC<QUEEN)then
    PC:=(PMEMORY[PC])mod PAWN
  else PC:=PC+1;
  CK:=CK+2
end;

```

```

        end;
procedure PBIOZ;
begin
    if (BIO=0) then PC:=(PMEMORY[PC]) mod PAWN
    else PC:=PC+1;
    CK:=CK+2
end;
procedure PBLEZ;
begin
    if (ACC=0) or (ACC>=QUEEN) then
        PC:=(PMEMORY[PC]) mod PAWN
    else PC:=PC+1;
    CK:=CK+2
end;
procedure PBLZ;
begin
    if ACC>=QUEEN then PC:=(PMEMORY[PC]) mod PAWN
    else PC:=PC+1;
    CK:=CK+2
end;
procedure PBNZ;
begin
    if ACC<>0 then
        PC:=(PMEMORY[PC]) mod PAWN
    else PC:=PC+1;
    CK:=CK+2
end;
procedure PBV;
begin
    if (OVFL=1) then
        begin
            PC:=(PMEMORY[PC]) mod PAWN;
            OVFL:=0
        end
    else PC:=PC+1;
    CK:=CK+2
end;
procedure PBZ;
begin
    if (ACC=0) then PC:=(PMEMORY[PC]) mod PAWN
    else PC:=PC+1;
    CK:=CK+2
end;
procedure PCALA;
var
    K : integer;
begin
    for K:=3 downto 1 do
        STACK[K]:=STACK[K-1];
    STACK[0]:=PC;
    PC:=ACC mod PAWN;
    CK:=CK+2
end;

```

```

        end;
Procedure PCALL;
var
    K : integer;
begin
    for K:=3 downto 1 do
        STACK[K]:=STACK[K-1];
    STACK[0]:=PC+1;
    PC:=PMEMORY[PC];
    CK:=CK+2;
    end;
Procedure PDINT;
begin
    INTM:=1;
    CK:=CK+1;
end;
Procedure PDMOV;
begin
    OPRND;
    DLOC:=DLOC+1;
    if (DLOC<144) then
        DMEMORY[DLOC]:=SCRATCH
    else DERROR;
    CK:=CK+1;
end;
Procedure PEINT;
begin
    INTM:=0; DELAY:=true;
    CK:=CK+1;
end;
Procedure PIN;
var
    X : char;
    K,D : integer;
begin
    OPRND;
    D:=IRH mod 8;
    if D=0 then
        begin
            B:=0;
            for K:=1 to 4 do
                begin
                    READ(INPUT,X);
                    if X in ['0'..'9','A'..'F']
                        then X:=X
                    else BADINST;
                    B:=B*16+HEXTOINT(X);
                end;
            READLN(INPUT);
            DMEMORY[DLOC]:=B;
        end
    else DMEMORY[DLOC]:=IMPORTED;
end;

```

```

        CK:=CK+2
    end;
procedure PLAC;
    var
        D : integer;
    begin
        OPRND;
        if(SCRATCH>=KNIGHT)then
            SCRATCH:=KING+SCRATCH-ROOK;
        D:=IRH mod 16;
        while D>0 do
            begin
                SCRATCH:=(SCRATCH*2)mod KING;
                D:=D-1
            end;
        ACC:=SCRATCH;
        CK:=CK+1
    end;
procedure PLACK;
    begin
        ACC:=IRL;
        CK:=CK+1
    end;
procedure PLAR;
    begin
        OPRND;
        if((IRH mod 8)div 2=0)then
            begin
                if(IRH mod 8=0)then AR0:=SCRATCH
                else AR1:=SCRATCH
            end
        else BADINST;
        CK:=CK+1
    end;
procedure PLARK;
    begin
        if((IRH mod 8)< 2 )then
            if(IRH mod 2=0)then AR0:=IRL
            else AR1:=IRL
        else BADINST;
        CK:=CK+1
    end;
procedure PLARP;
    begin
        if((IRL div 2)=64)then ARP:=IRL mod 2
        else BADINST;
        CK:=CK+1
    end;
procedure PLDP;
    begin
        OPRND;
        DP:=SCRATCH mod 2;
    end;

```

```

        CK:=CK+1
    end;
Procedure PLDPK;
    begin
        if((IRL div 2)=0)then DP:=IRL mod 2
        else BADINST;
        CK:=CK+1
    end;
Procedure PLST;
    begin
        OPRND;
        OVFL:=(SCRATCH div KNIGHT)mod 2;
        OVM:=(SCRATCH div 16384)mod 2;
        ARP:=(SCRATCH div 256)mod 2;
        DP:=SCRATCH mod 2;
        CK:=CK+1
    end;
Procedure PLT;
    begin
        OPRND;
        TR:=SCRATCH;
        CK:=CK+1
    end;
Procedure PLTA;
    begin
        OPRND;
        TR:=SCRATCH;
        SCRATCH:=PR;
        TEMP:=(ACC+SCRATCH)mod KING;
        PSGNA;
        CK:=CK+1
    end;
Procedure PLTD;
    begin
        OPRND;
        TR:=SCRATCH;
        SCRATCH:=PR;
        TEMP:=(ACC+SCRATCH)mod KING;
        PSGNA;
        DMEMORY[DLOC+1]:=DMEMORY[DLOC];
        CK:=CK+1
    end;
Procedure PMAR;
    begin
        OPRND;
        CK:=CK+1
    end;
Procedure PMPY;
    var
        P,Q :boolean;
    begin
        P:=true; Q:=true;

```



```

OPRND;
if (TR=KNIGHT) and (SCRATCH=KNIGHT) then
    PR:=QUEEN+QUEEN div 2
else
    begin
        if (SCRATCH>=KNIGHT) then
            begin
                P:=false;
                SCRATCH:=ROOK-SCRATCH
            end;
        TEMP:=TR;
        if (TEMP>=KNIGHT) then
            begin
                Q:=false;
                TEMP:=ROOK-TEMP
            end;
        PR:=TEMP*SCRATCH;
        if (P<>Q) then
            PR:=KING-PR
        end;
        DELAY:=true; CK:=CK+1
    end;
procedure PMPYK;
var
    P,Q : boolean;
begin
    P:=true; Q:=true;
    SCRATCH:=IR mod 8192;
    if (SCRATCH>=PAWN) then
        begin
            SCRATCH:=SCRATCH+57344;
            P:=false;
            SCRATCH:=ROOK-SCRATCH
        end;
    TEMP:=TR;
    if (TEMP > KNIGHT) then
        begin
            Q:=false;
            TEMP:=(ROOK-TEMP)
        end;
    PR:=TEMP*SCRATCH;
    if ((P and (not Q)) or ((not P) and Q)) then
        PR:=KING-PR;
    DELAY:=true; CK:=CK+1
end;
procedure PNOP;
begin
    CK:=CK+1
end;
procedure POR;
var
    K,D : integer;

```

```

begin
  OPRND;
  TEMP:=ACC mod ROOK;
  ACC:=0;
  D:=ROOK;
  for K:=1 to 16 do
    begin
      ACC:=ACC*2; D:=D div 2;
      if((SCRATCH div D)mod 2=1)or
        ((TEMP div D)mod 2=1)then
        ACC :=ACC+1
      end;
      CK:=CK+1
    end;
end;
Procedure POUT;
var
  X : char;
  D : integer;
begin
  OPRND;
  D:=IRH mod 8;
  OUTPORT[D]:=SCRATCH;
  if D=0 then
    begin
      B:=SCRATCH;
      WRITE(OUTPUT,INTTOHEX(B div 4096),
        INTTOHEX((B div 256)mod 16),
        INTTOHEX((B div 16)mod 16),
        INTTOHEX(B mod 16));
      WRITELN(OUTPUT)
    end;
    CK:=CK+2
  end;
end;
Procedure PPAC;
begin
  ACC:=PR;
  CK :=CK+1
end;
Procedure PPOP;
var
  K : integer;
begin
  ACC:=STACK[0];
  for K:=1 to 3 do
    STACK[K-1]:=STACK[K];
  CK :=CK+2
end;
end;
Procedure PFUSH;
var
  K : integer;
begin

```

```

        for K:=3 downto 1 do
            STACK[K]:=STACK[K-1];
        STACK[0]:=ACC mod PAWN;
        CK:=CK+2
    end;
Procedure PRET;
    var
        K : integer;
    begin
        PC:=STACK[0];
        for K:=1 to 3 do
            STACK [K-1]:=STACK[K];
        CK:=CK+2
    end;
Procedure PROVM;
    begin
        OVM:=0;
        CK :=CK+1
    end;
Procedure PSACH;
    var
        D : integer;
    begin
        OPRND;
        D:=IRH mod 8;
        if (D=0) or (D=1) or (D=4) then
            begin
                SCRATCH:=ACC;
                while (D>0) do
                    begin
                        SCRATCH:=(SCRATCH*2) mod KING;
                        D:=D-1
                    end;
                SCRATCH:=SCRATCH div ROOK;
                DMEMORY[DLOC]:=SCRATCH;
            end
        else BADINST;
        CK:=CK+1
    end;
Procedure PSACL;
    begin
        OPRND;
        DMEMORY[DLOC] :=ACC mod ROOK;
        CK:=CK+1
    end;
Procedure PSAR;
    begin
        OPRND;
        if (IRH mod 2)=0 then DMEMORY[DLOC]:=ARO
        else DMEMORY[DLOC]:=AR1;
        CK:=CK+1
    end;
end;

```

```

Procedure PSQVM;
begin
    OVM:=1;
    CK:=CK+1
end;

Procedure PSPAC;
begin
    SCRATCH:=KING-PR;
    TEMP:=(ACC+SCRATCH)mod KING;
    PSGNA;
    CK:=CK+1
end;

Procedure PSST;
begin
    if( IRL mod 128)=0 then
        begin
            if IRL<16 then DLOC:=128+IRL
            else BADINST
            end
        else OPRND;
        DMEMORY[DLOC]:=OVFL*KNIGHT+OVM*16384
        +ARF*256+DP;
        CK:=CK+1
    end;

Procedure PSUB;
var
    D : integer;
begin
    OPRND;
    if SCRATCH>=KNIGHT then
        SCRATCH:=KING+SCRATCH-ROOK;
    D:=IRH mod 16;
    while D>0 do
        begin
            SCRATCH:=(SCRATCH*2)mod KING;
            D:=D-1
        end;
    SCRATCH:=KING-SCRATCH;
    TEMP:=(ACC+SCRATCH)mod KING;
    PSGNA;
    CK:=CK+1
end;

Procedure PSUBC;
begin
    OPRND;
    if SCRATCH>=KNIGHT then
        SCRATCH:=KING+SCRATCH-ROOK;
    SCRATCH:=(SCRATCH*KNIGHT)mod KING;
    SCRATCH:=KING-SCRATCH;
    TEMP:=(ACC+SCRATCH)mod KING;
    if(TEMP div ROOK)< KNIGHT then
        ACC:=(TEMP*2+1)mod KING

```

```

        else ACC:=(ACC*2)mod KING;
        CK:=CK+1
    end;
Procedure PSUBH;
begin
    OPRND;
    SCRATCH:=SCRATCH*ROOK;
    SCRATCH:=KING-SCRATCH;
    TEMP:=(ACC+SCRATCH)mod KING;
    PSGNA;
    CK:=CK+1
end;
Procedure PSUBS;
begin
    OPRND;
    SCRATCH:=KING-SCRATCH;
    TEMP:=(ACC+SCRATCH)mod KING;
    PSGNA;
    CK:=CK+1
end;
Procedure PTBLR;
begin
    OPRND;
    STACK[3]:=STACK[2];
    PLOC:=ACC mod PAWN;
    DMEMORY[DLOC]:=PMEMORY[PLOC];
    CK:=CK+3
end;
Procedure PTBLW;
begin
    OPRND;  STACK[3]:=STACK[2];
    PLOC:=ACC mod PAWN;
    PMEMORY[PLOC]:=DMEMORY[DLOC];
    CK:=CK+3
end;
Procedure PXOR;
var
    P,Q :boolean;
    K,D :integer;
begin
    OPRND;
    TEMP:=ACC mod ROOK;
    ACC:=0;  D:=ROOK;
    for K:=1 to 16 do
        begin
            ACC:=ACC*2;  D:=D div 2;
            if(SCRATCH div D)mod 2=1 then P:=true
            else P:=false;
            if(TEMP div D)mod 2=1 then Q:=true
            else Q:=false;
            if(P<>Q)then ACC:=ACC+1
        end;
    end;
end;

```

```

        CK:=CK+1
    end;
Procedure PZAC;
    begin
        ACC:=0;
        CK:=CK+1
    end;
Procedure PZALH;
    begin
        OPRND;
        ACC:=SCRATCH*ROOK;
        CK:=CK+1
    end;
Procedure PZALS;
    begin
        OPRND;
        ACC:=SCRATCH;
        CK:=CK+1
    end;
Procedure PMISC;(IRH common,inspect IRL)
    var
        K : integer;
    begin
        K:=IRL mod 128;
        if K>29 then BADINST;
        if (K>2)and(K<8)then BADINST;
        if (K mod 28>16)then BADINST;
        case K of
            0 : PNOP;
            1 : PDINT;
            2 : PEINT;
            8 : PABS;
            9 : PZAC;
            10 : PROVMM;
            11 : PSDVM;
            12 : PCALA;
            13 : PRET;
            14 : PPAC;
            15 : PAPAC;
            16 : PSPAC;
            28 : PPUSH;
            29 : PPOP
        end
    end;
(EXECUTE BEGINS)
begin
    IR:=PMEMORY[PC];
    IRH:=IR div 256;
    IRL:=IR mod 256;
    PREVPC:=PC;
    PC:=PC+1;
    ERRFLAG:=false;

```

```
DELAY:=false;
if TRFLG then PTRACE;
case IRH of
  0 : PADD;
  1 : PADD;
  2 : PADD;
  3 : PADD;
  4 : PADD;
  5 : PADD;
  6 : PADD;
  7 : PADD;
  8 : PADD;
  9 : PADD;
 10 : PADD;
 11 : PADD;
 12 : PADD;
 13 : PADD;
 14 : PADD;
 15 : PADD;
 16 : PSUB;
 17 : PSUB;
 18 : PSUB;
 19 : PSUB;
 20 : PSUB;
 21 : PSUB;
 22 : PSUB;
 23 : PSUB;
 24 : PSUB;
 25 : PSUB;
 26 : PSUB;
 27 : PSUB;
 28 : PSUB;
 29 : PSUB;
 30 : PSUB;
 31 : PSUB;
 32 : PLAC;
 33 : PLAC;
 34 : PLAC;
 35 : PLAC;
 36 : PLAC;
 37 : PLAC;
 38 : PLAC;
 39 : PLAC;
 40 : PLAC;
 41 : PLAC;
 42 : PLAC;
 43 : PLAC;
 44 : PLAC;
 45 : PLAC;
 46 : PLAC;
 47 : PLAC;
 48 : PSAR;
```

49 : PSAR#
56 : PLAR#
57 : PLAR#
64 : PIN#
65 : PIN#
66 : PIN#
67 : PIN#
68 : PIN#
69 : PIN#
70 : PIN#
71 : PIN#
72 : POUT#
73 : POUT#
74 : POUT#
75 : POUT#
76 : POUT#
77 : POUT#
78 : POUT#
79 : POUT#
80 : PSACL#
88,89 : PSACH#
92 : PSACH#
96 : PADDH#
97 : PADD#
98 : PSUBH#
99 : PSUBS#
100 : PSUBC#
101 : PZALH#
102 : PZALS#
103 : PTBLR#
104 : PMAR#
105 : PDMOV#
106 : PLT#
107 : PLTD#
108 : PLTA#
109 : PMPY#
110 : PLDPK#
111 : PLDP#
112 : PLARK#
113 : PLARK#
120 : PXOR#
121 : PAND#
122 : POR#
123 : PLST#
124 : PSST#
125 : PTBLW#
126 : PLACK#
127 : PMISC#
128 : PMPYK#
129 : PMPYK#
130 : PMPYK#
131 : PMPYK#


```

132 : PMPYK;
133 : PMPYK;
134 : PMPYK;
135 : PMPYK;
136 : PMPYK;
137 : PMPYK;
138 : PMPYK;
139 : PMPYK;
140 : PMPYK;
141 : PMPYK;
142 : PMPYK;
143 : PMPYK;
144 : PMPYK;
145 : PMPYK;
146 : PMPYK;
147 : PMPYK;
148 : PMPYK;
149 : PMPYK;
150 : PMPYK;
151 : PMPYK;
152 : PMPYK;
153 : PMPYK;
154 : PMPYK;
155 : PMPYK;
156 : PMPYK;
157 : PMPYK;
158 : PMPYK;
159 : PMPYK;
244 : PBANZ;
245 : PBV;
246 : PBIOZ;
248 : PCALL;
249 : PB;
250 : PBLZ;
251 : PBLEZ;
252 : PBGZ;
253 : PBGEZ;
254 : PBNZ;
255 : PBZ;
      others : BADINST
end;
if INTRFLG then PINTERRUPT;
if FLGTR then TRACE
end{END EXECUTE};
procedure GO { execute in automatic mode };
begin
  repeat EXECUTE
  until BRKPT[PC] or ERRFLAG;
  if BRKPT[PC] then
    begin
      WRITELN(TTY);
      WRITELN(TTY, 'Break-at-user-pc: ',

```



```

BRKTABLECKJ.PLACE:=PLOC;
BRKPTCPLOCJ:=true
end
else
begin
WRITELN(TTY);
WRITELN(TTY,
'Brket-table full')
end
end
end
else ERROR;
GETSYM
end
end ( setbrk );
procedure INITBRK ( initialize break-point table );
var
K : integer;
begin
for K:=0 to 4095 do BRKPTCKJ:=false;
for K:=0 to PRED(NBRK)do BRKTABLECKJ.FLAG:=false
end ( initbrk );
procedure REMBRK ( remove break-points );
var
PLOC : 0..4095;
K : integer;
FLAG : boolean;
begin
GETSYM;
if SYM=LNEND then INITBRK;
while SYM<>LNEND do
begin
if SYM=NUMBER then
begin
PLOC:=B mod PAWN;
BRKSRCH(PLOC,K,FLAG);
if FLAG then
begin
BRKTABLECKJ.FLAG:=false;
BRKPTCPLOCJ:=false
end
end
else ERROR;
GETSYM
end;
end ( rembrk );
procedure LSTBRK ( list all break-points );
var
PLOC : 0..4095;
K : integer;
begin
for K:=0 to PRED(NBRK)do

```

```

        if BRKTABLECKJ.FLAG then
            begin
                PLOC:=BRKTABLECKJ.PLACE;
                WRITELN(TTY,
                    INTTOHEX(PLOC div 256),
                    INTTOHEX((PLOC div 16)mod 16),
                    INTTOHEX(PLOC mod 16))
            end
        end { 1stbrk };
Procedure CMDTABLE;{command table}
begin
    TABLEC1J.CMD:='BRK      ' ;
    TABLEC2J.CMD:='DEPDATA ' ;
    TABLEC3J.CMD:='DEPIPORT' ;
    TABLEC4J.CMD:='DEPPROG ' ;
    TABLEC5J.CMD:='DEPREG  ' ;
    TABLEC6J.CMD:='DISDATA ' ;
    TABLEC7J.CMD:='DISOPORT' ;
    TABLEC8J.CMD:='DISPROG ' ;
    TABLEC9J.CMD:='DISREG  ' ;
    TABLEC10J.CMD:='EX      ' ;
    TABLEC11J.CMD:='EXIT    ' ;
    TABLEC12J.CMD:='GO      ' ;
    TABLEC13J.CMD:='INTR    ' ;
    TABLEC14J.CMD:='LOAD    ' ;
    TABLEC15J.CMD:='LSTBRK  ' ;
    TABLEC16J.CMD:='PLOT    ' ;
    TABLEC17J.CMD:='REMBRK  ' ;
    TABLEC18J.CMD:='REMTRACE' ;
    TABLEC19J.CMD:='RESET   ' ;
    TABLEC20J.CMD:='SETTRACE'
end;
Procedure CMDPROC;{command processor}
var
    DONE :boolean ;
    FIRST, LAST, MIDDLE : integer ;
begin
    DONE := false ; COMMAND := A ;
    FIRST := 1 ; LAST := 20 ;
    while ((LAST >= FIRST) and (not DONE)) do
        begin
            MIDDLE := (FIRST+LAST) div 2 ;
            if COMMAND > TABLECMIDDLEJ.CMD then
                FIRST := MIDDLE + 1
            else
                if COMMAND < TABLECMIDDLEJ.CMD then
                    LAST := MIDDLE - 1
                else
                    if COMMAND=TABLECMIDDLEJ.CMD then
                        DONE := true
                    end ;
                end ;
            if DONE then

```

۱۱۷۱۷۱۷۱۷

d